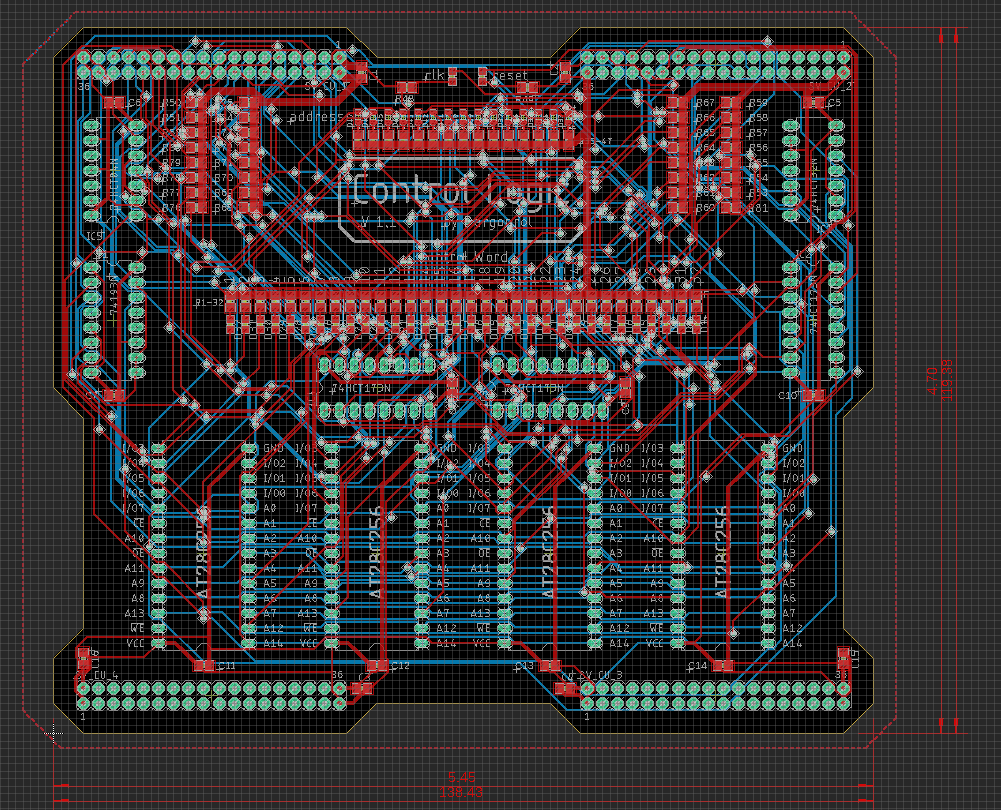
2023

**Bach Alexander**

Jugend forscht

25.3.2023



„Zu verstehen, ist die Essenz, um zu erschaffen.“

**Dragon-4 | Ein auf Logikebene selbst entwickelter Prozessor**

# Kurzfassung

Mein Ziel ist es die fundamentale Funktionsweise eines Prozessors zu verstehen. Schon oft habe ich gröbere Erklärvideos gesehen, die grundlegende Zusammenhänge darstellen wie zum Beispiel den Aufbau eines Prozessors. Doch all diese Lehrinhalte haben eines gemeinsam: Sie veranschaulichen nur einen groben Einblick in die komplexe Welt der Computer. So will ich nicht nur grundlegende Prinzipien verstehen, sondern mir einen so tiefgreifenden Einblick in die Thematik schaffen, dass ich selbst in der Lage wäre einen Prozessor von Grund auf selbst zu entwickeln. Und genau das tat ich. In diesem Projekt, konzentriert sich Jahrelange Liebe für Technik sowie stundenlanges Forschen und Entwickeln, für das Erschaffen eines eigenen Prozessors.

Inhalt

[Kurzfassung 1](#_Toc130655357)

[1. Einleitung 3](#_Toc130655358)

[1.1 Ursprünge 3](#_Toc130655359)

[1.2 Ein Traum wird Realität 3](#_Toc130655360)

[2. Vorgehens- und Funktionsweise 4](#_Toc130655361)

[2.1 Entwürfe, Planung, Anfänge 4](#_Toc130655362)

[2.1.1 Dragon-1 4](#_Toc130655363)

[2.1.2 Dragon-2 4](#_Toc130655364)

[2.1.3 Dragon-3, Pipeline 5](#_Toc130655365)

[2.1.4 Dragon-4 5](#_Toc130655366)

[2.2 Vorgehensweise 6](#_Toc130655367)

[2.2.1 Simulation 6](#_Toc130655368)

[2.2.2 ROM-Datengenerator 7](#_Toc130655369)

[2.2.3 PCB design 7](#_Toc130655370)

[2.2.4 Zusammenbau und Programmieren 8](#_Toc130655371)

[2.3 Funktionsweise 9](#_Toc130655372)

[2.3.1 Clock 9](#_Toc130655373)

[2.3.2 Register 9](#_Toc130655374)

[2.3.3 PC (Program Counter) 9](#_Toc130655375)

[2.3.4 MAR (Memory Address Register) 9](#_Toc130655376)

[2.3.5 RAM 10](#_Toc130655377)

[2.3.6 Control Logic (CU, CL) 10](#_Toc130655378)

[2.3.7 ALU 10](#_Toc130655379)

[2.3.8 IR (Instruction Register) 10](#_Toc130655380)

[3. Ergebnisse 11](#_Toc130655381)

[3.1 Probleme 11](#_Toc130655382)

[3.1.1 Pipeline 11](#_Toc130655383)

[3.1.2 Simulation vs. Realität 11](#_Toc130655384)

[3.1.3 Routing Probleme 11](#_Toc130655385)

[3.1.4 „Fehler sind Menschlich“ 12](#_Toc130655386)

[3.2 Leistung 12](#_Toc130655387)

[4. Ergebnisdiskussion 13](#_Toc130655388)

[5. Zusammenfassung 13](#_Toc130655389)

[6. Quellenverzeichnis 14](#_Toc130655390)

[7. Unterstützungsleistung 14](#_Toc130655391)

# 1. Einleitung

## 1.1 Ursprünge

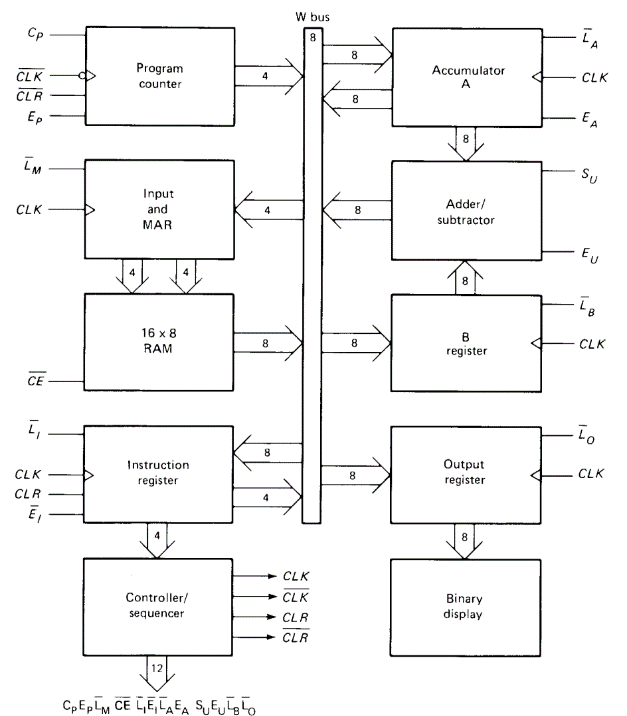
Seit jungen Jahren bin ich ein technikbegeisterter Mensch. Schon damals wurde ich mit kleinen Bastelkits oder Technikmuseen begeistert. Mein Opa brachte mir viel allgemeines Wissen bei und ich fing an meine eigenen Projekte zu realisieren. Doch den Einstieg in Digitaltechnik machte ich tatsächlich durch ein Computerspiel namens Minecraft. So simpel es doch ausschaut ist es möglich, ganze Computer in diesem Spiel zu kreieren. An Inspiration mangelte es nicht. Auf YouTube sah ich dauernd Menschen, welche Taschenrechner oder einfache Spiele erschaffen hatten. Nicht lange danach fing ich selbst an mich mit dieser Thematik zu beschäftigen und machte mich relativ schnell vertraut mit Logikgattern. Logikgatter sind die ultimative Basis jeder Digitaltechnik. Auch wenn man sie weiter in z.B. Transistoren unterteilen kann, bilden Logikgatter die wahre Essenz und absolute Grundlage. In Minecraft dauert es nicht lange, bis man an die Grenzen des Möglichen stößt, weswegen sich die Ausweitung auf eine Logiksimulations-software wie *Logisim* nicht lange vor sich warten ließ. Zum ersten Mal war es mir möglich, alle meine Ideen umzusetzen, ohne Geld in teures Material zu investieren. Mein Wissen erweiterte ich, selbst heute noch, mit kostenlosen Inhalten wie Videos auf YouTube. Wenn man lange genug sucht, findet man äußerst lehrreichen Inhalt, der nur darauf warten gesehen zu werden. Durch Hilfe von außerhalb erlangte ich auch Zugriff auf Bücher wie *The Art of Electronics* von Paul Horowitz und Winfield Hill, sowie *Digital Computer Electronics* von Albert P. Malvino und Jerald A. Brown. Mehrere Jahre experimentiere ich so herum und erweiterte mein Horizont. Erst Letztes Jahr entfachte in mir die Flamme, einen Prozessor zu entwickeln und in der Realität umzusetzen.

## 1.2 Ein Traum wird Realität

Der Wendepunkt meiner Geschichte wurde von nur einer einzigen Person inszeniert. Mein Projektbetreuer Benjamin Brück war es, der mich zu Jugend forscht brachte. Erst jetzt war es möglich das kostenintensive Hobby in die Realität umzusetzen. Die Deadline, die dadurch entstand, brachte natürlich einerseits Unmut, doch im Wesentlichen ist genau das der Grund, warum ich überhaupt Fortschritte gemacht habe. Man kann sich plötzlich keine drei Jahre mehr Zeitnehmen, sondern wird aktiv dazu gebracht, weiterzumachen. Einen Monat vor der Anmeldung begann ich schließlich intensiv zu forschen, verwarf mehrmals meine Konzepte und brachte schließlich eine teilfertige Version ins Leben. Vor allem James Sharman, ein Youtuber, welcher einen sog. *Pipelined processor[[1]](#footnote-1)* entwickelt hat, löste einen unglaublichen Motivationsschub hervor. Zunächst wollte ich seinen Prozessor aus Zwecken des Verstehens lediglich nachbauen, doch die Teilnahme an Jugend forscht stellte sich dem Quer und das aus gutem Grund. Für meine privaten Ziele, des Verstehens hätte ein einfaches Nachbauen ausgereicht, doch durch die Teilnahme an Jugend forscht wollte ich etwas Eigenes auf die Beine stellen. Einfach etwas nachbauen kann jeder, auch wenn das Nachbauen und das tatsächliche Verstehen zwei unterschiedliche Dinge sind. So nahm ich mir stattdessen Sharmans Werk als Vorbild und baute darauf meinen eigenen Prozessor. Die Architektur[[2]](#footnote-2) mag ähnlich sein, doch meine bis zu dem Punkt entwickelten Architekturen wurden restlos übertroffen. So musste ich einen Kompromiss eingehen. Ich nahm seine Architektur als Vorbild, veränderte sie und baute dann darauf auf. Ich wollte mich so weit wie möglich von ihm entfernen, doch die Genialität seiner Architektur beibehalten. So ist der Kern meines CPUs[[3]](#footnote-3) essenziell anders. Jedes Modul wurde von mir selbst entwickelt, simuliert, geplant und schließlich gebaut. Ich habe immense Zeit und stundenlanges Kopfzerbrechen in dieses Projekt investiert, um mein Ziel, das tiefgründige Verstehen eines Prozessors, zu erreichen.

# 2. Vorgehens- und Funktionsweise

## 2.1 Entwürfe, Planung, Anfänge

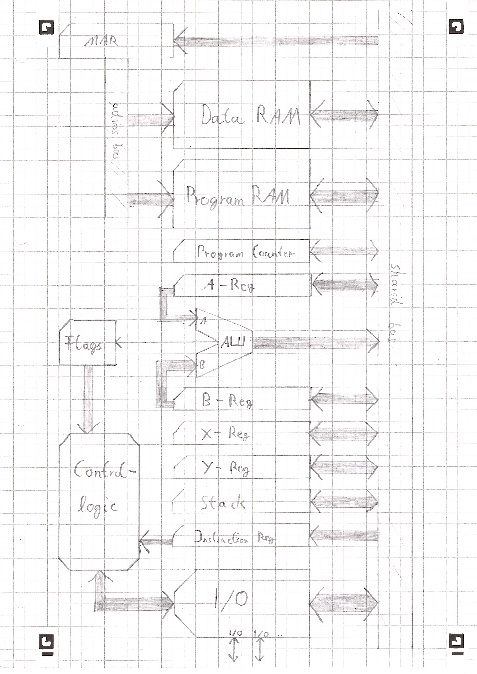
Zunächst begann alles mit dem SAP-1. SAP steht für Simple as Possible. Das ist Englisch und bedeutet so einfach wie möglich. Der SAP-1 ist eine Prozessor Architektur, vorgestellt von Albert P. Malvino in seinem Buch *Digital Computer Electronics* mit dem Ziel einem die Grundlagen eines Prozessors nahezubringen. Der SAP-1 ist die simpelste Form eines Turing vollständigen Prozessors. Turing vollständig bedeutet die universelle Programmierbarkeit eines Systems benannt nach Alan Turing. Im Endeffekt ist das ein System, welches in der Lage ist, sich selbst zu emulieren oder jede beliebige Berechnung zu tätigen, wenn sie unendlich Speicherplatz zur Verfügung hätte und die Berechnungszeit endlich ist. Den SAP-1 nachzubauen war mein erstes Ziel und der erste Schritt zu einem näheren Verständnis eines Prozessors. Alles weitere Wissen baut auf diesem Verständnis auf. Aber im Grunde gibt es eine Zentrale Einheit, die sog. *Control Logic*. Diese ist zuständig für die Erhaltung der Ordnung im System. Das macht sie, indem sie durch *control lines* Signale an die jeweiligen Module sendet. Das wiederum tut sie basierend auf *eng. Instructions*, Befehle, welche sie vom RAM[[4]](#footnote-4) bekommt.

### 2.1.1 Dragon-1

Der SAP-1 war mein Einstieg in Prozessoren. Einmal verstanden, kann man von da aus weiter seinen Weg gehen und komplexere Themen anstreben. Der Dragon-1, der erste Entwurf, war im Prinzip nichts anderes als der SAP-1, mit ein paar erweiterten Funktionen. U.a. konnte die ALU[[5]](#footnote-5) mehr logische Operationen, die die Ursprünge ALU des SAP-1 nicht konnte. Außerdem hatte der Dragon-1 mehr Register[[6]](#footnote-6) und vor allem ein Display. Mein Ziel war es damals, ein Display anzuschließen, bei dem sich jeder einzelne Pixel steuern lässt. Auch wenn meine Pläne wahrscheinlich in der Theorie funktioniert hätten, wären die Möglichkeiten sehr begrenzt und ineffizient gewesen. Auch wenn ich ihn schon beinahe fertig simuliert hatte, stufte ich den Dragon-1 äußerst unpraktisch ein und verwarf ihn.

Abbildung | SAP-1 Architektur

### 2.1.2 Dragon-2

Relativ schnell setzte ich mich schließlich an den Dragon-2, die Weiterentwicklung des Dragon-1. Hier lag der Fokus vor allem auf I/O-Handling[[7]](#footnote-7). Der Dragon-2 hat dazu im Unterschied zum Dragon-1 eine vollständig selbst entwickelte Architektur und versucht Hardware-interrupts[[8]](#footnote-8) zu integrieren. Ich entwickelte von Grund auf selbst ein System, um Interrupts zu Implementieren. Einer der größten unterschiede war aber wohl das Trennen vom Daten- und Programmspeicher und das hinzufügen eines 16-Bit Adress-busses. So ähnelt die Architektur der Harvard Architektur. Der Dragon-1 hingegen basierte auf der Von-Neumann Architektur, die einen gemeinsamen Bus für Adressen und Daten verwendet. Zu guter Letzt finden wir noch einen sog. Stack, der z.B. für *function-calling* genutzt wird. Damit war der Dragon-2 eine wesentliche Weiterentwicklung gegenüber dem Dragon-1. Beim dem Dragon-2 wäre es wahrscheinlich auch geblieben, doch fand er nie seinen Weg vom Papier in die Simulation, denn es gibt zwei Sachen, die mich umgestimmt haben. Einmal, war ich mir sehr unsicher bezüglich des I/O-Handling Systems. Ich hatte nicht die Zeit und die Mittel, um dieses System zu testen. Ich konnte mir nie sicher sein, dass es auch wirklich funktionieren würde. Man beachte, dass wir hier vor dem Zeitpunkt der Anmeldung an Jugendforscht reden. Zweitens entdeckte ich James Sharman, ein YouTuber, der seinen eigenen Prozessor nach dem Pipeline-Prinzip entwickelte.

### 2.1.3 Dragon-3, Pipeline

Eine Pipeline ist eine andere Art der Steuerungseinheit. Im Gegensatz zu *multicycle* Prozessoren werden hier mehrere Instruktionen gleichzeitig bearbeitet. Das führt dazu, dass sich die Zeit, die der Prozessor benötigt, um einen Befehl auszuführen, effektiv auf Dauer eines Taktes der Taktfrequenz verringert. Bei einem *multicycle* Prozessor benötigt eine *Instruction* meist mehrere Taktzyklen. Daher der Begriff *multicycle*. Mit der Pipeline liegt der größte Unterschied des Dragon-3 zum Dragon-2 aber in der Anzahl der Busse oder einfach gesagt in der Architektur generell. Diese ist wie bereits erwähnt stark inspiriert von James Sharman. Anstatt eines MARs[[9]](#footnote-9) haben wir einen *16-Bit Adress-Bus*, auf dem mehrere *Adress-Register* ihre Daten ablegen können. Außerdem ist der Haupt-Bus (*main bus*), auf dem Daten wie Ergebnisse von Berechnungen übertragen werden getrennt von dem Speicher-Bus (*memory bus*), sodass sich gleichzeitig Daten sowie auch neue Instruktionen bewegen können. Zu guter Letzt kann jedes Register Daten an die ALU mit separaten ALU-Bussen (*ALU-Left, ALU-Right*) senden. Das alles hat den Vorteil von deutlich mehr Flexibilität und Parallelisierung von Prozessen. Um I/O zu unterstützen, muss man lediglich ein Gerät an den Haupt-Bus anschließen und durch etwa einen I/O-Handler steuern lassen. Theoretisch ginge das auch direkt über die Steuereinheit (CU oder CL), die Anzahl von verfügbaren Steuerleitungen ist aber begrenzt. Auch der Dragon-3 war nicht weit entfernt von der fertigen Simulation, nichtsdestotrotz gibt es einige Punkte, weshalb ich nicht bei dem Dragon-3 geblieben bin. Der Hauptgrund ist wohl, die starke Ähnlichkeit zu James Sharmans CPU. Ein anderer Grund ist die wesentlich höhere Komplexität einer Pipeline. Zwar habe ich das Konzept einer Pipeline verstanden, jedoch würde es zu viel Zeit beanspruchen, eine eigene zu entwickeln, wenn man im Hinterkopf hat, dass wir uns hier schon unmittelbar nach der Anmeldung bei Jugend forscht befinden.

Abbildung | Selbst gezeichnete Architektur des Dragon-2

### 2.1.4 Dragon-4

So sind wir heute bei dem Dragon-4. Die Architektur basiert auf die des Dragon-3, also den von Sharman entworfenen Design, doch das Pipeline-Konzept, welches mich doch sehr inspiriert hat, wurde verworfen und wesentliche Änderungen an der Architektur vorgenommen.

Unter anderem wurde die Anzahl der 16-Bit Register auf drei reduziert und vor allem eine direkte Möglichkeit des RAMs Daten auf den Haupt-Bus aufzulegen erschaffen. Gleichzeitig wurde die Anzahl an GPRs[[10]](#footnote-10) erhöht und die Bus-Bridge, die in der von Sharmans Ursprünglichen Architektur den Haupt Bus und den Speicherbus verbunden hat, herausgenommen. Diese ist nämlich keine Notwendigkeit und wird durch das Abschaffen der Pipeline sowie eine direkte Verbindung des RAMs zum Haupt Bus auch nicht mehr benötigt.

Abbildung | Dragon-4 Architektur

## 2.2 Vorgehensweise

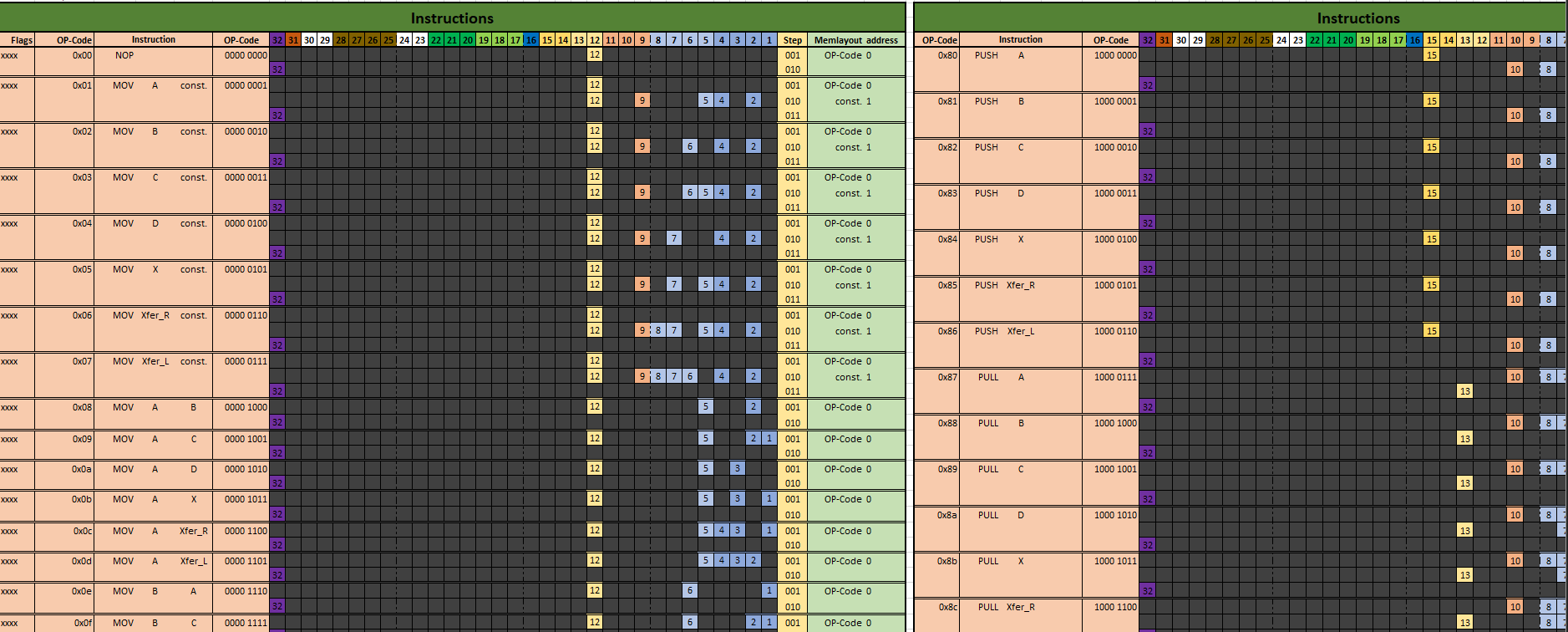
Zunächst einmal muss man das Grundgerüst, die System Architektur bilden, um anzufangen einen Prozessor zu entwickeln. Da mir diese durch den Dragon-3 bereits klar war, fing ich direkt damit an und nutze das von Microsoft für Windows verfügbare Programm Whiteboard (siehe Abb. 3). Dort ist das Grundgerüst entstanden. Natürlich muss einem vorher klar sein, was man erreichen möchte und was der Prozessor alles können soll, um darauf seine Architektur aufzubauen. Wichtig ist mir eine breite Anzahl an Funktionen, die der CPU leisten kann, sowie einfache Ein- und Ausgabe Möglichkeiten. Natürlich sollte er auch nicht zu langsam sein, um ein Minimalmaß an Nutzbarkeit zu erlangen. Für die Nutzbarkeit war vor allem ein breites Maß an Instruktionen notwendig. Jede Prozessorarchitektur hat seine eigene Programmiersprache, mit eigenen Befehlen, mit dem sie arbeiten kann, die Assemblersprache. Die Assemblersprache steht direkt vor der Maschinensprache mit Einsen und Nullen und besteht aus einfachen Begriffen z.B. MOV A B. Intel Prozessoren haben eine andere Syntax als AMD, da sie direkt vom Prozessortyp abhängig ist. Deshalb gibt es für meinen Prozessor keine fertige Sprache, sondern muss erst entwickelt werden. Dazu fertigte ich eine große Excel-Tabelle mit allen verfügbaren, von mir definierten Befehlen an. Da der Dragon-4 ein 8-Bit Prozessor ist, unterstützt er genau 256 verschiedene Befehle. Als Inspiration habe ich mir dazu das reale Instruction-Set des x86 Prozessors angeschaut.

Abbildung | Ausschnitt des Instruction Sets

### 2.2.1 Simulation

Da nun alle Grundeigenschaften gegeben waren, konnte ich mit der Entwicklung der einzelnen Module anfangen. Das allerdings im realen Leben zu tun ist kostspielig und aufwendig. Deshalb verwendete ich zunächst eine Logik-Simulationssoftware namens *Logisim. Logisim* ist ein großartiges Werkzeug, um Logikbausteine zu simulieren und zu testen. So habe ich sie zum Beispiel genutzt, um zu experimentieren welche Kombination von Logik Gattern ich benötige, um das gewünschte Ergebnis zu erhalten. Das hat mir das aufwendige Aufbauen und Verkabeln von realen Bauteilen erspart und den Prozess um einiges Beschleunigt. Schließlich habe ich den gesamten Prozessor simuliert, um sicherzugehen, dass er überhaupt funktionieren würde. Dabei sind einem viele Logik- und Denkfehler aufgefallen, die teilweise zu neuen Ideenanstöße geführt haben. Das alles, die Entwicklung des Prozessors in der Software, sowie das Ausmerzen von Fehlern und experimentieren beschreibt einen großen Teil der von mir investierten Ein Bild, das Diagramm, Plan enthält.

Automatisch generierte BeschreibungZeit in das Projekt.

Abbildung | Ausschnitt der Simulation des Dragon - 4

### 2.2.2 ROM-Datengenerator

Um den CPU vollständig in der Logiksoftware zu testen, muss natürlich auch die Steuereinheit vollständig sein. Das Problem, was sich hier aber auftut, ist, dass die Steuereinheit aus ROMs[[11]](#footnote-11) besteht, die den gigantischen Dekodierer ersetzen, der ansonsten von Nöten wäre. Das Interessante an einen ROM ist, dass man mit Speicherchips jegliche kombinierbare Logik ersetzen kann. Das mache ich mir hier zunutze, um nicht einen übermäßig großen Dekodierer bauen zu müssen. Dieser würde ansonsten jegliche Dimensionen sprengen. Nun müssen dazu aber vorher die ROMs mit Daten gefüllt werden. Die hier verwendeten ROMs haben 32000 mal 8 Bit Speicherplatz. Also 32000 Zeilen mit jeweils 8 Bit oder 1 Byte an Daten. Die Steuereinheit nutzt vier Stück von diesen. Die Daten also von Hand einzutragen wäre nicht nur wahnsinnig, sondern auch noch eine große Zeitverschwendung. Dazu schrieb ich ein Programm in Python, welches für mich alle 32000 Zeilen an Daten füllt. Ich muss lediglich die 256 Instruktionen einmal von Hand in dem Programm definieren.

Abbildung | Ausschnitt des "ROM-Datengenerators"

### 2.2.3 PCB design

Nachdem die Simulation einwandfrei funktionierte, wurde es Zeit, das Projekt in die Realität umzusetzen. Mir schien die Option, das Ganze auf Platinen zu bauen, effizienter und einfacher, als es mit handgefertigten Pantinen oder Breadboards zu versuchen. So arbeitete ich mit der PCB[[12]](#footnote-12)-Design Software Eagle um Schaltpläne des Dragon-4 zu erstellen und die PCBs zu designen. Dieser Prozess beanspruchte ebenfalls einen großen Teil meiner gesamt Investierten Zeit. Man beachte allerdings, dass „gesamt Investierte Zeit“ sich nur auf den Zeitraum von der Anmeldung bei Jugend forscht bis zur Fertigstellung des Projektes bezieht. Der Dragon-1 Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibungist bereits zwei Jahre vorher entstanden. Über diese zwei Jahre verteilt wurde immens viel Zeit mit Recherche und Weiterbildung zu diesem Thema verbracht, sodass es schwierig ist, die gesamt Investierte Zeit zu schätzen.

Bevor ich anfangen konnte, die PCBs zu designen oder die Schaltungen zu erstellen, musste ich mir erst über das Layout im Klaren werden. Hier stellte ich mir die Frage, wie soll der Prozessor später aussehen? Ich entschied mir für ein modulares Design, da Fehler unausweichlich sind und so diese einfacher zu korrigieren sind und ein modulares Design am meisten Flexibilität bietet. Ich kann einfach ein fehlerhaftes Modul herausnehmen und mit einem neuen ersetzen. Die tatsächlichen Maße weichen durch Weiterentwicklung während des PCB-Designs etwas ab, im Großen und Ganzen spiegelt Abbildung 7 aber sehr genau die tatsächliche Größe wider.

Abbildung | Physisches Layout vom Dragon-4

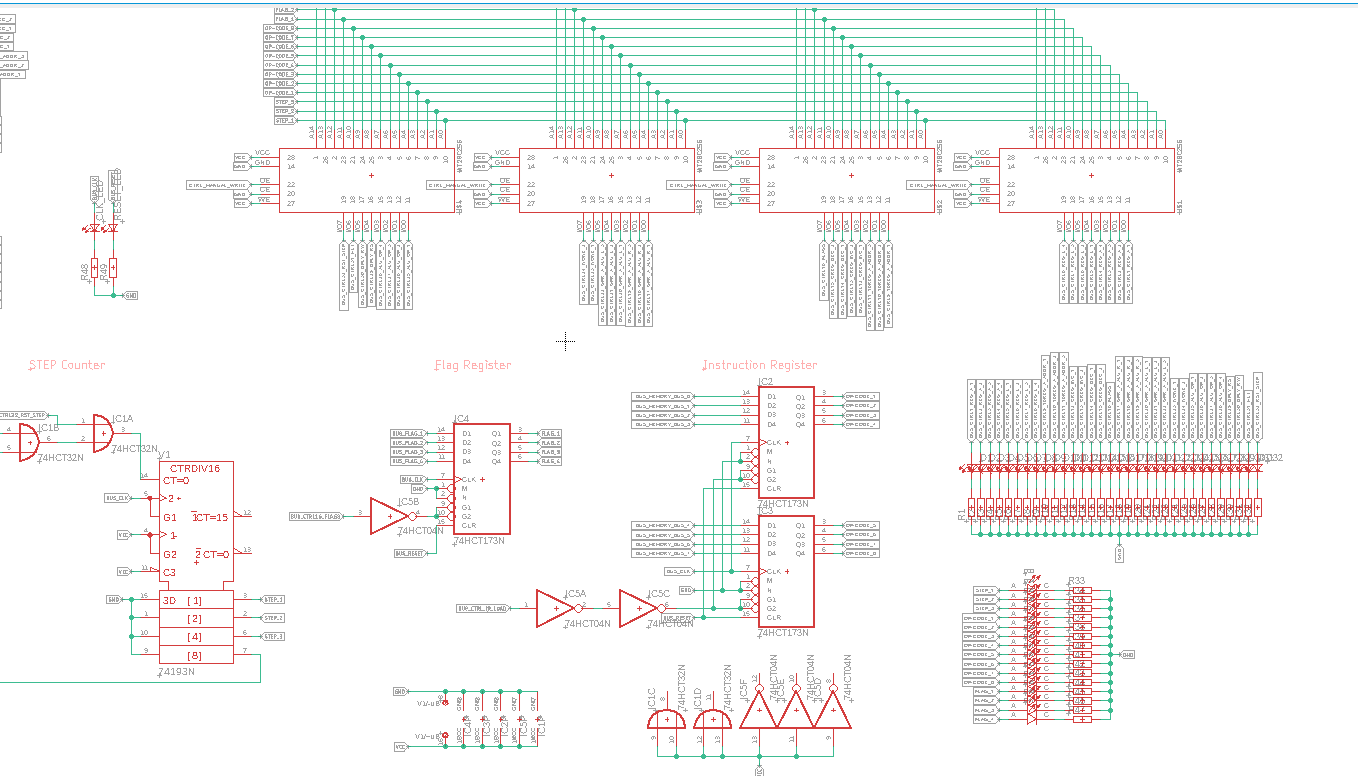
Nachdem das Layout stand, konnte ich anfangen, die Module zu erstellen. Zunächst der Schaltplan, dann das PCB. Hier galt es allerdings einiges zu beachten. Zunächst einmal muss man schauen, wie man die Bauteile in der Simulation in einen Schaltplan überträgt, denn die Simulation bietet nur idealisierte Bauteile. Ich musste also durch Recherche Bauteile finden, die dieselbe Funktion haben wie in der Simulation. Außerdem musste ich mich entscheiden, welche LEDs, welche Widerstände, welche Kondensatoren, welche Verbinder und noch vieles mehr ich nutzen möchte. Falls es diese dann nicht in dem Programm existieren, muss ich sie selbst erstellen. Zu beachten gibt es hier auch Dinge wie z.B. Lieferzeiten von Bauteilen und Verfügbarkeit. Ist die Schaltung einmal fertig gestellt, kann es zum PCB-Design übergehen. Hier musste ich mich erst mit allgemeinen Designregeln beschäftigen wie zum Beispiel, wann und wo man einen Kondensator platziert, und lernen mit dem Programm umzugehen. Da ich ein modulares Design verwende, muss es noch eine Grundplatte geben, die alles verbindet, das Motherboard. Nachdem alle Teile fertig designt waren, mussten diese lediglich von einem Platinen Hersteller gefertigt werden.

Abbildung | Ausschnitt der Kontroll-Logik

### 2.2.4 Zusammenbau und Programmieren

Nach Anfertigen der Teileliste und deren bestellen. Steht nur noch das Zusammenfügen vor einem. Nach Löten von über 6000 Pins, wochenlanges verbessern von Fehlern und debuggen ist der Prozessor fertig. Doch äußerst viel kann er in dem Zustand noch nicht, denn es müssen noch die ROMs, welche sich in der Steuerungseinheit befinden, beschrieben werden. Danach muss man lediglich ein Programm in den RAM laden und schon konnte der Prozessor seiner Bestimmung nachgehen.

## 2.3 Funktionsweise

Die Funktionsweise in einem Umfang zu erklären, welche nicht die Limitierungen dieser Facharbeit sprengen würde, ist schwierig, weshalb ich mich hier relativ kurzfasse.

Um die Funktionsweise des Dragon-4 zu verstehen, ist ein gewisses Grundwissen notwendig. Dieses erläutert man am besten anhand des SAP-1. Immerhin ist der SAP-1 so einfach wie nur möglich aufgebaut. Um den folgenden Erklärungen besser zu folgen, ist auf der nächsten Seite wieder die Architektur des SAP-1, nur dieses Mal etwas größer, abgebildet.

### 2.3.1 Clock

Die Clock ist wohl das simpelste Bauteil eines jeden Prozessors. Die Clock ist dazu zuständig, nach einem definierten Zeitintervall periodisch ein Signal auszusenden. Dieses wird dazu verwendet den gesamten Prozessor zu synchronisieren, damit jedes Bauteil fehlerfrei miteinander kommunizieren kann. Die Frequenzen erreichen in modernen Prozessoren mehr als . Im Dragon-4 ist die Frequenz durch seine Größe und verwendeten Bauteile sehr begrenzt. Dadurch, dass das Clock-Modul noch nicht fertiggestellt wurde, wurden noch keine Hochgeschwindigkeitsversuche durchgeführt, ich rechne aber mit einer theoretischen maximalen Frequenz von . Die bisher höchste getestete Frequenz liegt bei .

### 2.3.2 Register

Das Register, ein fester Bestandteil eines jeden Prozessors, tut nichts anderes als in meinem Fall 1-2 Byte an Daten für spätere Wiederverwendung zu speichern. Anders als bei dem RAM, haben wir hier nur **eine** Zeile mit Daten. Wenn das Register ein Signal zum Speichern empfängt, wird bei dem nächsten Clock-Zyklus die Daten, die sich am angrenzenden Datenbus befinden gespeichert. Nach empfangen eines Ausgabe-Befehls wirft das Register seine Daten wieder auf den Datenbus.

### 2.3.3 PC (Program Counter)

Der Programmzähler ist nichts anders als ein Register, welches die Fähigkeit hat, binär zu zählen. Auf Kommando erhöht es den Wert in seinem Register um eins. Das passiert genau nach jeder Instruktion. So könnte man ihn auch Instruction Pointer nennen. Denn er zählt im Grunde nur mit, bei welchem Punkt wir uns im Programm befinden. Er zeigt auf die Adresse im RAM, bei der die nächste Instruction gespeichert ist, damit diese ausgeführt werden kann.

Abbildung | SAP-1 Architektur

### 2.3.4 MAR (Memory Address Register)

Das MAR ist ein weiteres Register, welches die Aktuelle Adresse des RAMs speichert. Im SAP-1 wird der Inhalt des *program counters* in das MAR verschoben, um die aktuelle Adresse zu ändern. In meinem Prozessor gibt es direkt kein MAR. Hier wird der Inhalt des *program counters* so lange auf einen Adressbus ausgegeben, bis sie nicht mehr benötigt wird. Der Adressbus ist direkt mit dem RAM verbunden.

### 2.3.5 RAM

Der RAM ist einer der zentralen Komponenten. Er beinhaltet das Programm, welches wir ausführen möchten und so die Instruktionen, die dem Prozessor sagen, was er zu tun hat. RAMs können mehrere Hunderttausende Speicherbänke mit Datensätze beinhalten. Welche Speicherbank gerade ausgewählt ist, wird als Adresse bezeichnet. In meinem Fall besitzt der RAM Speicherbänke mit jeweils einem Byte, also Daten. Der RAM kann seine Daten entweder auf einen BUS ausgeben, oder direkt Steuereinheit überreichen.

Der RAM des Dragon-4 ist gleichzeitig das Stack. Das Stack wird zum *function-calling* verwendet. Die Besonderheit, dass der Stack in dem RAM integriert ist, ist dass der Speicherplatz für beides, Stack und Programm variable ist. Wenn man nicht aufpasst, könnte man den einen Bereich mit dem anderen überschreiben, andererseits hat es den Vorteil, dass wenn man mehr RAM benötigt, einfach mehr nutzten kann.

### 2.3.6 Control Logic (CU, CL)

Die Steuereinheit ist das Herz eines jeden Prozessors. Ohne sie, wäre er nicht funktionsfähig. Die Steuereinheit nimmt sich eine Instruktion, welche sie von dem Instruktions-Register oder dem RAM bekommt und sagt dem ganzen Rest des CPUs, was er zu tun hat. Das macht sie mithilfe eines riesigen Dekodierers, der für jede Instruction einen Befehlssatz das sog. *Control Word* ausgibt.

### 2.3.7 ALU

Viele bezeichnen die ALU, Arithmetisch Logische Einheit als das Gehirn eines Prozessors, doch ohne die anderen Komponenten ist sie sehr nutzlos. Wie der Name bereits hergibt, nimmt sie ein oder mehrere Werte von z.B. dem A und B Register (siehe Abbildung 9) oder von einem Bus, wie in dem Dragon-4 und führt mit diesen Werten Arithmetische Operationen wie plus- oder minusrechnen oder logische sowie bitwise Operationen wie AND oder *bit shifting* durch.

Abbildung | gesamter Prozessor mit unfertigen Modulen

### 2.3.8 IR (Instruction Register)

Das Instruktions-Register speichert die derzeitig bearbeitete Instruktion, damit sie von der der Kontroll-Logik genutzt werden kann. Im Dragon-4 ist dieses in der Steuereinheit implementiert.

# 3. Ergebnisse

## 3.1 Probleme

Einen Prozessor zu entwickeln ist keine einfache Aufgabe. Das zeigte sich immer wieder während der Entwicklung, sowie auch während des Zusammenbauens. Fangen wir bei dem Dragon-3 an, der noch auf einer Pipeline basierte.

### 3.1.1 Pipeline

Die Pipeline ist ein geniales Konzept, welches die Nutzung von der internen Logik maximal optimiert. Doch mit der Genialität wächst auch die Komplexität. Eine Pipeline beinhaltet mehrere Stages, bei der eine Instruktion jeweils unterschiedliche *Control-words* lossenden muss. Fehler führen zu einem nicht ausführen einer Instruktion oder das Fetchen[[13]](#footnote-13) einer nicht Instruktion. Herauszufinden welche Steuerleitung, in welcher Pipeline Stufe wann aktiviert werden muss, war sehr herausfordernd und hat einige Zeit beansprucht. Zu der ersten Ausführung eines Befehles, hat es sehr lange gedauert.

Die Entscheidung das Pipeline-Konzept zu verwerfen, hat alle Probleme des Dragon-3 hinfällig gemacht. Für die Komplexität einer Pipeline, war einfach nicht genügend Zeit vorhanden, nichtsdestotrotz habe ich das Konzept und die Funktionsweise einer Pipeline verstanden und zu verstehen sowie sich weiterzubilden ist wie schon oft beschrieben, meine Hauptmotivation.

### 3.1.2 Simulation vs. Realität

In der Simulation gibt es Probleme, die man in der Realität nicht hat, in der Realität gibt es Probleme, die man in der Simulation nicht hat. Vereinfacht gesagt, Simulation und Realität sind etwas komplett Unterschiedliches. Mit einer Simulation kann man schließlich eine Realität nur nachahmen. Das hat unausweichlich zu einigen, teilweise fatalen, Problemen geführt. Zunächst einmal sind die Bauteile, die man in der Logik-Software verwenden konnte, nur idealisierte Bauelemente, die so in der Realität nicht existieren. Das brachte mich dazu reale Äquivalente zu finden, doch ist es nun mal so, dass es nicht immer welche gibt. So musste ich kreativ werden und nach weiteren Bauteilen suchen, die diese ersetzen können. Dabei sind einige Änderungen Im Schaltplan im Vergleich zu der Simulation entstanden. Ein Beispiel wäre, dass fast alle Control Lines, invertiert wurden. Das war aufgrund der Bauweise der Chips. Diese Tatsache führte zu einem meiner größten Probleme, denn invertiere Control Lines bedeutete eine teilweise komplett unterschiedliche Logik. In Kombination mit den unterschiedlichen Bauteilen führte das dazu, dass die im Schaltplan verwendete Logik vollständig in meinem Kopf entstand und so viele Fehler enthielt. Erst einen Tag vor der Bestellung der Platinen fielen mir nochmal so viele Fehler auf, dass ich beschlossen habe, die Idealisierten Bauteile in der Simulation so anzupassen, dass sie der Realität entsprechen. So fand ich nochmals einige fatale Fehler, wie zum Beispiel das Anwenden falscher Logikgatter im RAM-Modul. Das hätte dazu geführt, dass dieses nicht ganz oder sogar gar nicht funktioniert hätte. Durch das realitätsnähere Simulieren kann ich mir zumindest etwas sicherer sein, dass mein Entwurf funktioniert.

### 3.1.3 Routing Probleme

Das ganze Verbessern der Fehler beinhaltet das Neurouten von den Leiterbahnen auf den Platinen. Auf kleineren Boards ist das kein Problem, doch auf größeren komplexeren, wie die ALU wird das sehr schnell sehr unübersichtlich und schwierig. Da auf solch einem Board teilweise über 300 Verbindungen getätigt werden müssen, benutzte ich einen Autorouter, der mir die Leiterbahnen automatisch nach einem Algorithmus verlegt. Man sollte lediglich gewisse Voreinstellungen wie Leiterbahndicke festlegen. So schön das auch klingen mag, bei solch hochkomplexen Platinen benötigt der Autorouter teilweise stundenlange Rechenzeit und manchmal versagt er sogar, da es schlicht weg einfach unmöglich sein kann, auf so engen Platz so viele routen zu verlegen. Als Lösung arrangierte ich Bauteile neu und stellte die Leiterbahndicke niedriger, wobei man hier einen Kompromiss eingehen muss, denn je dünner die Bahnen, desto weniger Ein Bild, das Text enthält.

Automatisch generierte BeschreibungStrom kann durch sie hindurchfließen. Die ALU erfolgreich zu routen hat dann schlussendlich mehrere Stunden gedauert.

Abbildung | fehlerhafte Platine

### 3.1.4 „Fehler sind Menschlich“

Nach dem Bestellen und zusammenbauen, passierte erst einmal nichts. Natürlich mussten sich Fehler, die man nicht voraussehen konnte, einschleichen, denn es wäre leichtsinnig gewesen, zu vermuten, dass alles auf Anhieb funktionieren würde. Trotzdem beansprucht das Debuggen von Problemen immense Zeit und kann sehr frustrierend sein. Doch dadurch lernt man, was ein wichtiger Prozess ist. Das Debuggen ist ein Prozess, der mehrere Wochen in Anspruch genommen hat. Letztendlich musste ich sogar den größten Teil der Platinen nochmal neu erstellen.

## 3.2 Leistung

Die Frage, was der Dragon-4 leisten kann ist eine schwierige, da ich noch nicht die Möglichkeit hatte ihn auf seine Höchstleistung zu testen. Fragt man mich aber was der Dragon-4 alles so kann, antworte ich *alles*. Das liegt daran, dass der Dragon-4 nach Alan Turing Turing Vollständig ist. D.h. er ist in der Lage jede beliebe Berechnung zu tätigen. Das beinhaltet auch hoch moderne Grafiken. Hätte der Prozessor eine Grafikkarte, wäre er auch in der Lage, diese anzuzeigen. Nach meinen Hypothesen sollte er bis zu einer Geschwindigkeit von laufen können. Limitiert wird er von der Geschwindigkeit der ROM-chips, da diese am größten und komplexesten sind und so am meisten Zeit benötigen, bis eingehende Signale bearbeitet werden. Wenn eine Instruktion durchschnittlich ca. 5 Zyklen benötigt und wir eine maximale Taktfrequenz von haben, kommen wir auf:

Da eine Pipeline theoretisch pro Zyklus eine Instruction verarbeitet, ist diese in dem Fall doppelt so schnell. Realistisch würde ich allerdings die Geschwindigkeit meines Prozessors geringer schätzen.

Der Grund, weshalb moderne Prozessoren wesentlich schneller sind, ist da diese mikroskopisch klein sind, und so die Bauteile nah aneinander liegen. Außerdem verwenden Hersteller modernste Technologien, auf die ich keinen Zugriff habe.

**Allgemeine Statistiken:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Anzahl Instruktionen** | **durchschnittliche Länge der Instruktionen** | **Maximale Taktfrequenz** | **Speicherplatz** | **IPS (instructions per second)** | **Stromverbrauch** |
|  |
| 256 | 5 Zyklen | ca. 1,6 Mhz | 512 kB | 334 kIPS | ca. 1 Watt |  |

# 4. Ergebnisdiskussion

Allgemein bin ich sehr zufrieden mit meinem Prozessor, doch Pläne habe ich noch viele mehr. Unter anderem wollte ich Ursprünglich eine Grafikkarte bauen, um die errechneten Ergebnisse zu visualisieren. Doch eine Grafikkarte ist kein einfaches Bauteil. Ich hätte wesentlich mehr Zeit benötigt, um eine Grafikkarte fertig zu stellen, vor allem da es für mich wichtig ist, dass ich keine fertigen Microcomputer für meinen CPU benutzte. Ich möchte alles von Grund auf selbst bauen. Das gleiche gilt für eine Soundkarte. I/O Handling war ein Ziel, welches ich Ursprünglich unbedingt implementiert haben wollte, doch auch hier war wegen der Vorbereitung auf die annähernden Abiturprüfungen im April wenig Zeit. Schließlich baute ich den Prozessor so, dass es theoretisch möglich wäre, einen I/O-handler anzuschließen, sollte ich in Zukunft Zeit und Lust finden, diesen zu bauen. Auch die Funktion durch eine Tastatur den Inhalt des RAMs zu verändern, ist momentan noch nicht möglich, da ich die dazu fehlende Hardware nicht rechtzeitig fertig stellen konnte. Trotzdem habe ich auch hier vorausgeplant und die Module so entwickelt, dass man diese Hardware einfach dazu stellen könnte. Bisher muss man also den RAM herausnehmen und sein Programm extern über einen Computer programmieren. Eine weitere Sache für die Zukunft wäre einen *assembler* zu schreiben. Bisher muss ich Programme direkt im Maschinencode programmieren, was sehr aufwendig ist.

Außerdem haben die fehlende Erfahrung in den diversen Programmen mir einige Schwierigkeiten und Frust bereitet, doch letztendlich habe ich nur dazugelernt. Sollte ich erneut einen Prozessor bauen oder diesen weiterentwickeln, weiß ich, wie. Vor allem das PCB-Design würde ich in Zukunft anders machen. Ich weiß zum Beispiel, dass man alle Bauteile nur auf die Oberfläche platzieren sollte, um sich eine Menge Arbeit und Kosten bei der Produktion zu sparen. Auch habe ich gelernt, dass es durch schlechte oder zu wenige Ground-Verbindungen zu Instabilität in den Chips kommen kann.

Rein theoretisch müsste es auch durch kleine Modifikationen möglich sein, das Pipeline-Konzept wieder einzuführen. Da der Prozessor sehr modular ist, kann man einfach Module austauschen. In dem Fall müsste man die gesamte Steuereinheit neu entwickeln. Das sind Dinge, die man sich in Zukunft vornehmen könnte.

# 5. Zusammenfassung

Trotz all den dutzenden Problemen bin ich sehr zufrieden. Dieses Projekt hat mir sehr viel Frust bereitet, aber auch umso mehr Erfahrung in allen Teilbereichen. Insgesamt hat es einen großen Spaß den Prozessor zu entwickeln und bauen, was das aller wichtigste ist.

Vor allem das zu demselben Zeitpunkt immer näher Rückende Abitur hat mir besonders Kopfschmerzen und immensen Zeitdruck veranstaltet. Nichtsdestotrotz war und ist meine Begeisterung für Technik und dem Forschen groß und hat schließlich dazu geführt, dass ich das Projekt rechtzeitig, nicht unbedingt abschließen, aber bis zu einen Zufriedenstellenden Punkt fertigstellen konnte. Mein Ziel, zu verstehen, habe ich erreicht. Ich verstehe nicht zu 50%, sondern zu 100%, wie zumindest eine solch simple Form eines Prozessors funktioniert. Zu verstehen, ist nicht nur das Ziel dieses einen Projektes, sondern das Ziel mehrere Jahre Begeisterung für Digitaltechnik. Man könnte sogar noch weiter gehen und sagen, es war Ziel, seit Anbeginn meiner Kindheit, seit Anbeginn meiner Sehnsucht und Liebe nach Technik. Schon als kleines Kind wollte ich wissen, wie die Welt funktioniert. Seither strebt mich das an weiter zu forschen, weiter zu entdecken, zu erfinden, zu leben. Technik ist Teil meines Lebens und ich bin dankbar dafür. Das Ziel im Engen Sinne, diesen Prozessor zu bauen, habe ich erreicht. Auch wenn manche Dinge nicht vollständig funktionieren, oder anders sind als erwartet, ist mein Ziel etwas selbst entwickeltes ins Leben zu rufen geglückt. Doch Ende ist hier noch lange nicht. Zu verstehen ist die Basis für alles Darauffolgende. Wenn man versteht, kann man kreieren, wenn man versteht, kann man weiterkommen. So will ich weiter verstehen, weiter nach Wissen streben, um immer höhere Horizonte zu erreichen. Ich bin dankbar für diese einzigartige Chance meinen Horizont zu erweitern und mir Wege für die Zukunft zu eröffnen. Dieses Projekt hat mir unglaublichen Spaß gemacht.

# 6. Quellenverzeichnis

* *Wikipedia, Turing Vollständigkeit, https://de.wikipedia.org/wiki/Turing-Vollständigkeit, zuletzt besucht am 12.02.2023*
* *Wikipedia, Instruction pipelining, https://en.wikipedia.org/wiki/Instruction\_pipelining, zuletzt besucht am 12.02.2023*
* *Wikipedia, Instructions per second*, https://en.wikipedia.org/wiki/Instructions\_per\_second*, zuletzt besucht am 25.03.2023*
* *Albert P. Malvino, Jerald A. Brown: „Digital Computer Electronics”, Kalifornien 1977*
* *Ben Eaters Video Reihe: „Building an 8-bit breadboard computer!”*
* *James Sharmans Video Reihe: „Making an 8 Bit pipelined CPU”*

# 7. Unterstützungsleistung

Ich möchte meinen herzlichsten Dank an allen Personen ausrichten, die mich bei der Planung bis zur Realisierung dieses Projektes unterstützt haben.

Besonderes Dankeschön an:

* **Wolfgang Schröder**, für Ideenfindung und Wissensbereicherung
* **Ben Eater**, für den Einstieg in das Themenfeld der Funktionsweise von Prozessoren durch seine Video-Reihe
* **James Sharman**, für die Umfangreiche Ideenfindung und Wissensbereicherung, sowie Inspiration und Erläuterungen von komplexen Konzepten wie der Pipeline durch seine Video-Reihe

Und ganz besonders an:

**Benjamin Brück**,

für das in Kontakt bringen mit Jugend forscht, der finanziellen Unterstützung, das Bereitstellen diverser Materialien, Werkzeugen und Räumen, das Korrekturlesen dieser Arbeit und die Hilfestellung bei diversen Fragen

1. Siehe 2.2.3 Dragon-3, Pipeline [↑](#footnote-ref-1)
2. Architektur, hier: Aufbau eines Prozessors [↑](#footnote-ref-2)
3. CPU = Eng. **C**entral **P**rocessing **U**nit | Prozessor [↑](#footnote-ref-3)
4. RAM = **R**andom **A**ccess **M**emory | veränderbarer Speicherplatz für Daten [↑](#footnote-ref-4)
5. **A**rithmetic **L**ogic **U**nit kurz ALU oder Arithmetisch Logische Einheit | zuständig für Logische und Arithmetische Berechnungen. [↑](#footnote-ref-5)
6. Register | zuständig für Temporäres Speichern von Daten [↑](#footnote-ref-6)
7. I/O = Input Output | I/O-Handling meint das Unterstützen von Geräten wie Tastatur und Display [↑](#footnote-ref-7)
8. Unterbrechung des laufenden Programmes für z.B. eine Tastatureingabe [↑](#footnote-ref-8)
9. MAR = **M**emory **A**ddress **R**egister | Speicherort für die Adresse des RAMs [↑](#footnote-ref-9)
10. GPR = **G**eneral **P**urpose **R**egister | Temporärer Speicherort für Daten [↑](#footnote-ref-10)
11. ROM = **R**ead **O**nly **M**emory | Speicherchip, von dem nur gelesen werden kann [↑](#footnote-ref-11)
12. PCB = **P**rinted **C**ircuit **B**oard | Platine [↑](#footnote-ref-12)
13. To fetch an instruction beschreibt den Prozess einen neuen Befehl in die Kontroll-Logik aufzunehmen, um ihn abzuarbeiten. [↑](#footnote-ref-13)