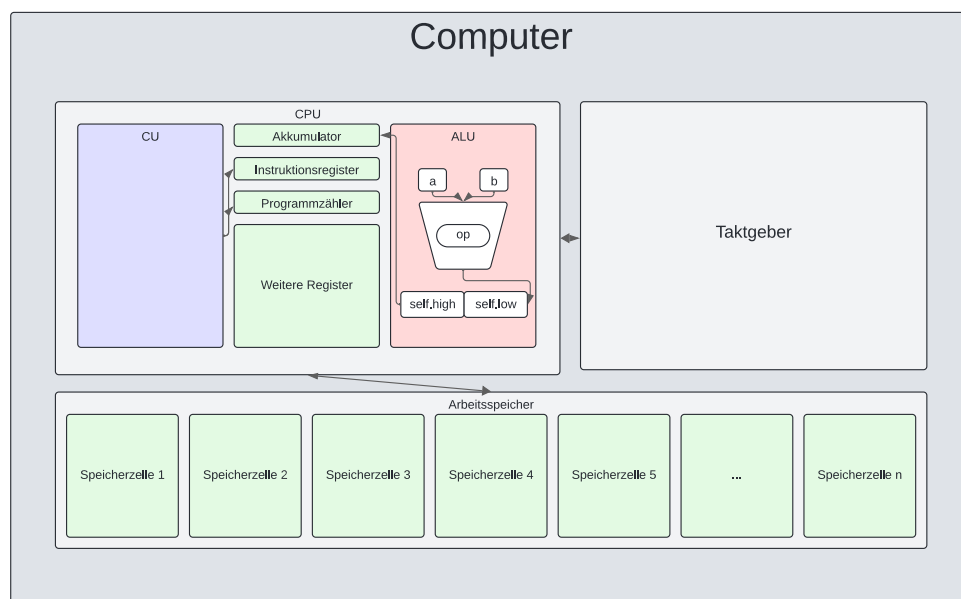




Maturaarbeit 2024 / 2025

Bern, 16. Oktober 2024

Computerarchitektur: CPU-Emulator in Python



Entwurf und Implementierung eines einfachen
Prozessor-Emulators zur Veranschaulichung von
Computerarchitektur

Malik Niederhauser, Abteilung MN, M25c

Betreut durch Thomas Jampen

Abstract	4
Vorwort	5
1 Einleitung.....	6
1.1 Zielsetzung.....	6
1.2 Relevanz	6
1.3 Überblick	6
2 Theoretische Grundlagen	8
2.1 Computerarchitektur	8
2.1.1 Von-Neumann-Architektur	8
2.1.2 Von-Neumann-Zyklus	9
2.1.3 Instruction Set Architecture (ISA)	10
2.1.4 Turing-Vollständigkeit	10
2.2 Emulatoren	10
2.3 Python als Programmiersprache	11
3 Produktidee und Anforderungen	12
3.1 Beschreibung des Produkts.....	12
3.2 Anforderungen	12
3.3 Befehlssatz des Emulators	13
3.3.1 Befehlstabelle	13
3.3.2 Legende	13
3.4 Klassenstruktur	14
3.4.1 Die Klasse CPU (Central Processing Unit).....	14
3.4.2 Die Klasse ALU (Arithmetic Logic Unit)	14
3.4.3 Die Klasse CU (Control Unit).....	15
3.4.4 Die Klasse RAM (Random Access Memory)	16
3.4.5 Die Klasse Clock.....	16
3.4.6 Die Klasse Computer	17
3.4.7 Die Klasse PixelDisplay	17
3.4.8 Die Klasse GUI (Graphic User Interface)	17

3.4.9 Die Klasse AppController	17
3.5 Programmlogik	18
3.6 Rolle des Assemblers	19
4 Prozessdokumentation	20
4.1 Entwicklungsprozess	20
4.2 Herausforderungen und Lösungen	21
5 Analyse und Bewertung	22
5.1 Bewertung des Produkts	22
5.2 Analyse	22
5.3 Reflexion	24
5.4 Ausblick	25
6 Fazit	26
6.1 Zusammenfassung	26
6.2 Abschliessende Gedanken	26
Verweise	27
Abbildungsverzeichnis	29
Programmier- und Schreibhilfen	30

Abstract

In dieser Arbeit wird ein auf einer eigenen ISA basierender CPU-Emulator in Python entwickelt. Das Ziel ist es, einen Emulator zu programmieren, der interne Prozesse eines Computers simulieren und visualisieren kann. Mithilfe eines Assemblers können anschliessend Programme für den Emulator geschrieben werden, um die Grenzen des Emulators in der Praxis zu finden.

Der Emulator kann einfache Programme ausführen. Dabei kann der Benutzer den internen Zustand der CPU sowie den Von-Neumann-Zyklus beobachten. Wegen dem kleinen Arbeitsspeicher, der begrenzten Leistung und dem Fehlen von Gleitkommazahlen hat der Emulator Schwierigkeiten beim Ausführen von komplexen oder modernen Anwendungen.

Der Emulator dient als Lernhilfe zur Veranschaulichung von Computerarchitektur, ist jedoch in Bezug auf Leistung begrenzt.

Vorwort

Schon am Kickoff-Event in der Aula war mir klar, dass das Thema meiner Maturaarbeit im Themenbereich der Informatik sein wird. Im ersten Jahr am Gymnasium habe ich angefangen mich mehr für Informatik zu interessieren. Im dritten Jahr habe ich Informatik als mein Ergänzungsfach gewählt und mich für das Freifach Informatik+ angemeldet. Am Anfang des dritten Jahres habe ich mich mehr mit Betriebssystemen auseinandergesetzt. Zuerst wollte ich für meine Maturaarbeit etwas im Themenbereich Betriebssystem machen, habe aber bemerkt, dass das nicht in den Rahmen der Maturaarbeit passt. Ich wollte gerne etwas machen, das ein tiefgründiges Verständnis in einem Bereich der Informatik verlangt. Auf YouTube bin ich auf mehrere Videos gestossen, in denen jemand einen Computer in Minecraft [1], Terraria und sogar Excel [2] gebaut hat. Es faszinierte mich, dass diese Computer, sofern sie Turing-vollständig sind, die Fähigkeit haben jede mögliche Berechnung und jedes mögliche Programm durchzuführen.

1 Einleitung

1.1 Zielsetzung

Mein Ziel ist es einen Computer in Python zu emulieren, der

1. Turing-vollständig ist.
2. programmierbar ist.
3. mit einer grafischen Benutzeroberfläche (GUI) bedient werden kann.
4. einen pixelbasierten Bildschirm hat.
5. eine eigene Befehlssatzarchitektur (ISA) hat.
6. als Hilfsmittel dienen kann, jemandem zu erklären, wie Computer funktionieren.

In der Theorie kann ein Turing-vollständiger Emulator jedes berechenbare Problem lösen. In der Praxis gibt es aber oft limitierende Faktoren, die das Lösen von komplexen Problemen erschweren. Ich will herausfinden, wozu ein selbst programmierter Emulator in der Praxis fähig ist.

1.2 Relevanz

Ein Programm, das einen Computer simuliert, wird natürlich nicht besser, schneller oder praktischer sein als es der Computer, auf dem das Programm läuft, schon ist. Was bringt also dieses Programm, wenn es nicht mehr als der Computer kann? In einem gewöhnlichen Computer kann man nicht sehen, was in welchen Registern gespeichert ist. Man sieht also nicht, was die CPU gerade macht oder was im Arbeitsspeicher gespeichert ist. Anders als ein herkömmlicher Computer kann ein Emulator den Zustand der CPU zu jederzeit ausgeben.

1.3 Überblick

Meine Arbeit ist in mehrere Abschnitte aufgeteilt, die von der Theorie über den Entwicklungsprozess zur Analyse des Endprodukts reichen.

Im theoretischen Teil erkläre ich verschiedene Konzepte der Computerarchitektur. Ich behandle die Themen der Befehlssatzarchitektur und Turing-Vollständigkeit sowie die Von-Neumann-Architektur und den Von-Neumann-Zyklus. Anschliessend erläutere ich, warum ich Python als Programmiersprache gewählt habe sowie was ein Emulator ist und wo Emulatoren eingesetzt werden.

Im praktischen Teil beschreibe ich das Produkt und erkläre, wie es funktioniert. Ausserdem zeige ich meinen Arbeitsprozess und die Wichtigkeit des Assemblers.

Im letzten Teil teste ich den Emulator und analysiere die Resultate. Anhand der Ergebnisse kann ich die Herausforderungen und Grenzen des Emulators bestimmen und potenzielle Erweiterungen für zukünftige Entwicklungen finden.

Die Arbeit endet mit einem Fazit, in dem die Ergebnisse reflektiert und die Bedeutung des Projekts zusammengefasst werden.

2 Theoretische Grundlagen

2.1 Computerarchitektur

2.1.1 Von-Neumann-Architektur

Ein einfacher Computer nach der Von-Neumann-Architektur besteht aus einem Arbeitsspeicher (Speicherwerk, RAM), einem Ein- und Ausgabewerk, einem Bussystem, einem Taktgeber und einem Prozessor (Central Processing Unit, CPU). Der Prozessor lässt sich noch in die Arithmetic Logic Unit (Rechenwerk, ALU), die Control Unit (Steuerwerk, CU) und verschiedene Register aufteilen.

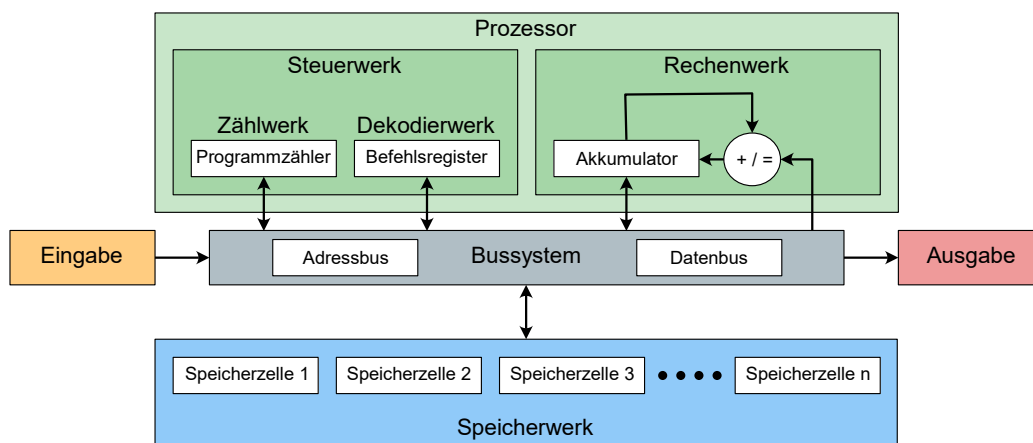


Abbildung 1: Von-Neumann-Architektur [3]

Der Taktgeber schickt wiederkehrende Signale an die Komponenten im Computer. Diese Signale sorgen dafür, dass die Teile im Computer synchron arbeiten. Der Taktgeber teilt den verschiedenen Komponenten also mit, wann sie ihre Aufgabe ausführen müssen.

Der Arbeitsspeicher speichert Daten, die von einem Programm benutzt werden, wie auch das Programm selbst. Der Arbeitsspeicher ist allerdings ein flüchtiger Speicher. Das heisst, dass wenn der Computer ausgeschaltet wird, alle im Arbeitsspeicher gespeicherten Daten verloren gehen.

Die CPU hat verschiedene Aufgaben. Sie ist für das Ausführen von Befehlen, die als Programm im Arbeitsspeicher gespeichert sind, zuständig. Auch Datenverarbeitung, also mathematische und logische Operationen (z.B. AND oder OR), gehört zu den Aufgaben der CPU. Die Resultate dieser Operationen werden in Registern gespeichert. Andere Komponenten im Computer werden von der CPU gesteuert. So sendet die CPU beispielsweise Signale an den Arbeitsspeicher, um zu steuern, dass Daten gelesen oder geschrieben werden.

Das Bussystem ermöglicht die Kommunikation zwischen den verschiedenen Komponenten. So können Daten, Adressen und Steuersignale ausgetauscht werden.

Zusätzlich haben Computer oft Ein- und Ausgabegeräte. Ein- und Ausgabegeräte werden benutzt, um Benutzereingaben zu erhalten, zum Beispiel durch Tastatur oder Maus, oder um etwas auszugeben, zum Beispiel Ton bei Lautsprechern oder ein Bild bei einem Monitor.

2.1.2 Von-Neumann-Zyklus

Der Von-Neumann-Zyklus besteht aus drei Schritten, die immer wieder wiederholt werden.

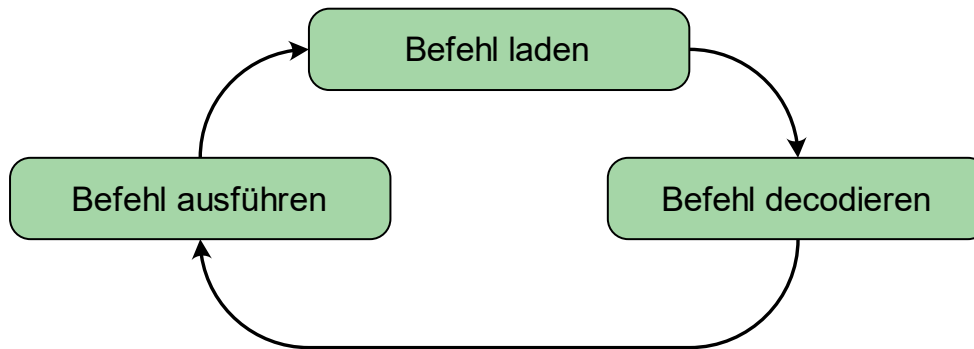


Abbildung 2: Von-Neumann-Zyklus [3]

Im ersten Schritt, Befehl laden (*Fetch*), wird der Befehl aus dem Arbeitsspeicher gelesen und in das Befehlsregister geschrieben. Die Adresse des Befehls steht im Programmzähler. Der Programmzähler wird erhöht und zeigt auf den nächsten Befehl.

Im zweiten Schritt, Befehl decodieren (*Decode*), wird der Befehl dekodiert. Befehle werden immer in Binär codiert und so im Arbeitsspeicher gespeichert. Die Art des Befehls (z.B. logische Operation oder Datenbewegung) und die nötigen Operanden werden in diesem Schritt identifiziert.

Im dritten Schritt, Befehl ausführen (*Execute*), kann der Befehl ausgeführt werden. Je nach Art des Befehls werden unterschiedliche Teile der CPU benötigt. Bei logischen Operationen wird beispielsweise die ALU benötigt während bei Datenbewegung die Register benötigt werden.

2.1.3 Instruction Set Architecture (ISA)

Die Befehlssatzarchitektur oder ISA ist eine Art Vorlage für die CPU. Die ISA definiert zum Beispiel die Instruktionen, die von der CPU ausgeführt werden können, und wie diese codiert werden. Die ISA beschreibt, was eine CPU tun kann, und die CPU-Hardware setzt um, wie es konkret gemacht wird. Unterschiedliche CPUs können also dieselbe ISA benutzen, sie aber auf eine andere Art und Weise implementieren. Intel- und AMD-Prozessoren benutzen beide die x86-ISA. Beide Prozessoren können also dasselbe, sind aber anders aufgebaut.

2.1.4 Turing-Vollständigkeit

Eine Maschine ist Turing-vollständig [4], wenn sie jedes berechenbare Problem lösen kann. Die einzigen Voraussetzungen sind genügend Speicherplatz und Zeit. Damit eine CPU Turing-vollständig sein kann, muss die ISA bestimmte Instruktionen umfassen. Die minimal notwendigen Instruktionen sind bedingte Sprünge, wie `jne` (*jump not equal*), und Speicherzugriffe, wie `store` und `load`. Überraschenderweise sind arithmetische und logische Operationen gar nicht zwingend nötig, machen das Lösen von Problemen aber viel unkomplizierter. Auch eine Instruktion wie `halt` braucht es nicht zwingend. Man könnte die CPU nach dem Lösen eines Problems in eine Endlosschleife versetzen. Die Benutzung einer `halt` Instruktion ist trotzdem praktisch, um Systemressourcen nach dem Ausführen eines Programms freizugeben.

2.2 Emulatoren

Ein Emulator ist ein Softwareprogramm, das ein anderes System, zum Beispiel einen alten Computer oder eine Spielkonsole, nachahmt. Programme, die für andere Hardware entwickelt wurden, können mithilfe eines Emulators ausgeführt werden.

Es gibt verschiedene Arten von Emulatoren. Retro-Gaming ist eine bekannte Anwendung von ihnen. Abgesehen davon werden sie auch beim Entwickeln und Testen von Software benutzt. Ausserdem kann alte Software, die nicht mehr auf modernen Betriebssystemen läuft, mithilfe von Emulatoren immer noch ausgeführt werden.

In der Forschung werden Emulatoren zum Beispiel zur Emulation von Quantencomputern benutzt. Forscher und interessierte können Quantenalgorithmen auf Emulatoren von Quantencomputern testen, ohne Zugang zu Quantencompu-

tern zu benötigen. Auch beim Testen von neuen Betriebssystemen werden Emulatoren benutzt, damit man das Betriebssystem auf mehreren Plattformen testen kann, ohne Zugang zu jeder Hardware zu haben.

Mit Emulatoren lässt sich das Verständnis von Hardware und Architekturen festigen. Sie können Lernenden die Funktionsweise von Computern näherbringen. Komplexe Prozesse innerhalb des Computers, die normalerweise nicht sichtbar sind, können mit Emulatoren simuliert und visualisiert werden.

2.3 Python als Programmiersprache

Für jemanden, der sich ein wenig mit Informatik oder Programmierung auskennt, mag die Wahl der Programmiersprache für dieses Projekt überraschend sein. Python ist im Vergleich zu anderen Sprachen wie C, C++ oder Rust bekanntlicherweise langsam. C, C++ und Rust bieten dem Programmierer mehr Möglichkeiten für Low-Level-Speicherverwaltung durch Funktionen wie `malloc` und `free`. Manuelle Speicherverwaltung und Zeiger können Performance und Speichereffizienz erheblich verbessern, was für ein Emulator natürlich von grossem Vorteil ist. Warum habe ich mich also für Python und gegen C, C++ und Rust entschieden? Die Gründe sind Risiken und Komplexität. Speicherlecks, Zugriffsverletzungen und schwer zu findende Bugs sind typische Risiken von C und C++. Rust bietet die Vorteile von Low-Level-Speicherverwaltung und minimiert dabei die Risiken. Rust gilt allerdings als schwierige Sprache aufgrund der steilen Lernkurve und der schwierigen Syntax. Da ich mit C, C++ und Rust zu Beginn des Projekts noch nicht so viel Erfahrung hatte, machte es nur Sinn Python als Programmiersprache für dieses Projekt zu wählen.

Ein weiterer Grund, der für Python spricht, ist, dass ich bereits Erfahrung mit Python Bibliotheken gesammelt habe, die für dieses Projekt nützlich sind. Für die Implementation des Emulators habe ich 7 Bibliotheken benutzt. `sys` [5], `pathlib` [6], `threading` [7], `tkinter` [8], `os` [9], `pygame` [10] und `time` [11].

3 Produktidee und Anforderungen

3.1 Beschreibung des Produkts

Das Produkt ist ein Python-Programm, welches beim Start eine grafische Benutzeroberfläche öffnet. Der Benutzer kann anschliessend den emulierten Computer steuern und für diesen geschriebene Programme darauf laufenlassen. Dabei hat der Benutzer die Wahl das Programm automatisch ausführen zu lassen oder manuell Taktzyklus für Taktzyklus durchs Programm zu gehen.

Aus dem Hauptfenster wird der emulierte Computer gesteuert. Zudem können weitere Fenster geöffnet werden. In einem Fenster für den simulierten Monitor wird das Pixeldisplay abgebildet, welches als grafische Ausgabe dient. Andere Fenster, wie das der CPU oder des Taktgebers, zeigen an, was diese Komponenten des Computers gerade tun, und visualisieren diese Prozesse.

3.2 Anforderungen

Für mich ist die grafische Benutzeroberfläche wichtig. Sie soll intuitiv sein, dem Benutzer einen Einblick in die Computer-Architektur geben und dem Benutzer helfen zu verstehen, wie ein Computer funktioniert. Dabei spielt mir die Leistung des Emulators keine grosse Rolle. Der Emulator ist definitiv nicht optimiert und könnte wahrscheinlich auch schneller laufen.

Das Programmieren eines Assemblers ist mir auch wichtig, weil er das Schreiben von Programmen für den Computer extrem vereinfacht.

Das ganze Programm soll dazu modular aufgebaut sein, sodass einzelne Komponenten des Computers aber auch das GUI und der Assembler unabhängig voneinander entwickelt werden können. Neue Features und Verbesserungen sollen einfach implementiert werden können.

3.3 Befehlssatz des Emulators

3.3.1 Befehlstabelle

Instruktion	Opcode	Op 1	Op 2	Op 3	Beschreibung
jmp #imd	00000000	Pad	Pad	Imd	imd -> pc
jeq #imd	00000001	Pad	Pad	Imd	Falls zero flag = 1, imd -> pc
jne #imd	00000010	Pad	Pad	Imd	Falls zero flag != 1, imd -> pc
inc [mem]	00000011	Pad	Pad	Mem	[mem] += 1
dec [mem]	00000100	Pad	Pad	Mem	[mem] -= 1
load #imd, reg	00000101	Reg 1	Pad	Imd	imd -> reg1
load [mem], reg	00000110	Reg 1	Pad	Mem	[mem] -> reg1
store reg, [mem]	00000111	Reg 1	Pad	Mem	reg1 -> [mem]
jmp reg	00001000	Reg 1	Pad	Pad	reg1 -> pc
jeq reg	00001001	Reg 1	Pad	Pad	Falls zero flag = 1, reg1 -> pc
jne reg	00001010	Reg 1	Pad	Pad	Falls zero flag != 1, reg1 -> pc
store reg1, reg2	00001011	Reg 1	Reg 2	Pad	reg2 -> [reg1]
move reg1, reg2	00001100	Reg 1	Reg 2	Pad	reg2 -> reg1
add reg1, reg2	00001101	Reg 1	Reg 2	Pad	reg2 + reg1 -> reg1
sub reg1, reg2	00001110	Reg 1	Reg 2	Pad	reg2 - reg1 -> reg1
mult reg1, reg2	00001111	Reg 1	Reg 2	Pad	reg2 * reg1 -> High in reg1, Low in reg2
div reg1, reg2	00010000	Reg 1	Reg 2	Pad	reg2 / reg1 -> Quotient in reg1, Rest in reg2
inc reg	00010001	Reg 1	Pad	Pad	reg1 += 1
dec reg	00010010	Reg 1	Pad	Pad	reg1 -= 1
and reg1, reg2	00010011	Reg 1	Reg 2	Pad	reg2 & reg1 -> reg1
or reg1, reg2	00010100	Reg 1	Reg 2	Pad	reg2 reg1 -> reg1
xor reg1, reg2	00010101	Reg 1	Reg 2	Pad	reg2 ^ reg1 -> reg1
not reg1	00010110	Reg 1	Pad	Pad	~reg1 -> reg1
rol reg1, #imd	00010111	Reg 1	Imd	Pad	reg1 <<< imd -> reg1
ror reg1, #imd	00011000	Reg 1	Imd	Pad	reg1 >> imd -> reg1
cmp reg1, reg2	00011001	Reg 1	Reg 2	Pad	reg2 - reg1 (ändert nur Flags)
shl reg1, #imd	00011010	Reg 1	Imd	Pad	reg1 << imd -> reg1
shr reg1, #imd	00011011	Reg 1	Imd	Pad	reg1 >> imd -> reg1
halt	11111111	Pad	Pad	Pad	Beendet das Programm

3.3.2 Legende

Imd	Direkter Wert
Mem	Speicheradresse
Pad	Padding
Pc	Programcounter (Programmzähler)
Reg1, Reg2	Register 1, Register 2

3.4 Klassenstruktur

3.4.1 Die Klasse CPU (Central Processing Unit)

Die CPU-Klasse lässt sich in drei Hauptteile aufteilen: Die ALU, die CU und die Register. Da die ALU und CU selbst grössere Komponenten sind, haben die beiden Einheiten ihre eigenen Klassen. Die Register der CPU werden als Attribut gespeichert und jeweils als Strings mit 16 beziehungsweise 32 Nullen initialisiert.

- `r0-r7`: Acht 16bit Allzweckregister.
- `pc` (*Program Counter*): Beinhaltet die Speicheradresse der nächsten Instruktion.
- `acc` (*Accumulator*): Speichert die Ergebnisse der ALU. (32 bit)
- `ir` (*Instruction Register*): Speichert den geholten Befehl (32 bit)
- `flags`: Speichert 16 flags, die den Zustand der CPU beschreiben.

Neben den oben genannten Registern gibt es auch noch den Stack Pointer (`sp`), den Base Pointer (`bp`), das Memory Address Register (`mar`) und das Memory Data Register (`mdr`).

Da ich zum Zeitpunkt der Abgabe noch keinen Stack implementiert habe, sind Stack Pointer und Base Pointer überflüssig. Der Stack Pointer würde die Speicheradresse des obersten Elements im Stack beinhalten und der Base Pointer würde auf die Basis des aktuellen Stack-Frames zeigen.

`mar` und `mdr` sind auch nicht zwingend notwendig, da mit den Instruktionen `store` und `load` irgendwelche beliebigen Register als Operanden angegeben werden können und die Daten direkt von einem Register in eine Speicherzelle im Arbeitsspeicher beziehungsweise von einer Speicherzelle im Arbeitsspeicher in ein Register geschrieben werden. Daten werden in echten Computern, im Gegensatz zu diesem Emulator, immer über das `mdr` transferiert.

5.4.2 Die Klasse ALU (Arithmetic Logic Unit)

Die ALU-Klasse hat zwei 16bit Register, damit, bei der Multiplikation zum Beispiel, Resultate bis zu $2^{32}-1$ gespeichert werden können, in dem die ersten 16bit in `self.high` und die anderen 16bit in `self.low` gespeichert werden. Die wichtigste Methode der ALU-Klasse ist

`execute(self, op, a, b, bit_length = 16)`. Mithilfe eines Match-Case Statements wird der `op` Parameter der dazugehörigen Methode der Operations-Klasse zugewiesen. Die Parameter `a`, `b` sind die Operanden der arithmetischen oder logischen Operation und der `bit_length` Parameter gibt an, mit

welcher Anzahl Bits gerechnet werden soll. Die Anzahl Bits muss spezifiziert werden, weil am Schluss jeder arithmetischen Operation das Resultat Modulo $2^{(\text{bit_length})}$ gerechnet werden muss, damit sich das Resultat noch mit der gewünschten Anzahl Bits darstellen lässt. Nach dem Ausführen einer Operation werden noch Flags gesetzt. Wenn eine Subtraktion beispielsweise null ergibt, wird die Zero-Flag zu eins gesetzt. Falls die Compare-Flag null ist, werden die beiden Register `self.high` und `self.low` in den Akkumulator geschrieben. Bei echten Computern gibt es auch noch andere Flags, wie zum Beispiel eine Overflow-Flag, welche zu eins gesetzt wird, wenn es einen Overflow gibt.

3.4.3 Die Klasse CU (Control Unit)

Die CU-Klasse hat drei Methoden, die den Von-Neumann-Zyklus simulieren.

Die erste Methode ist `fetch(self)`. Diese Methode geht im Arbeitsspeicher zum Index, der vom Program Counter spezifiziert ist, holt die nächsten 4 Byte aus dem Arbeitsspeicher und speichert diese ins Befehlsregister. Es werden immer 4 Bytes geholt, weil eine Instruktion 32 Bit, was 4 Bytes entspricht, lang ist. Anschliessend wird 4 zum Program Counter addiert, damit der Program Counter zur nächsten Instruktion zeigt.

Die zweite Methode ist `decode(self)`. Sie holt die Instruktion und spaltet sie auf. Die ersten 8 Bit stehen für den Opcode, also welche Instruktion aufgerufen wird. Danach gibt es zwei 4-Bit-Operanden und einen 16-Bit-Operanden.

Die dritte Methode ist `execute(self)`. Diese Methode funktioniert ähnlich wie die vorher genannte Methode der ALU. Das match-case-Statement entscheidet welche statische Methode der `Instructions`-Klasse ausgeführt wird. Falls für eine Instruktion eine arithmetische oder logische Operation ausgeführt werden muss, wird diese Aufgabe der ALU weitergegeben. Das Resultat kann anschliessend aus dem Akkumulator gelesen werden.

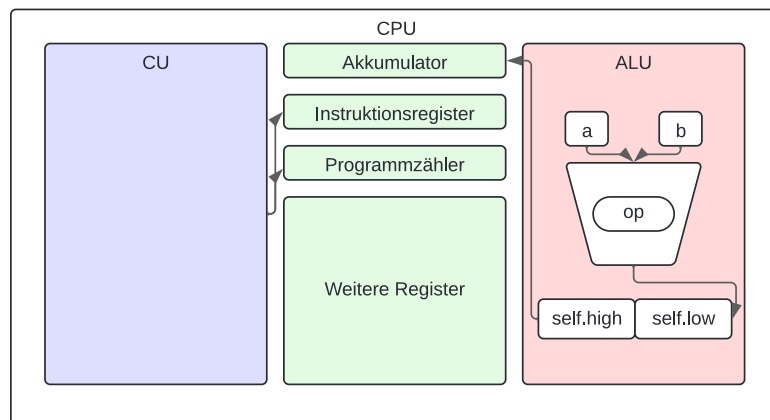


Abbildung 3: Aufbau der CPU

3.4.4 Die Klasse RAM (Random Access Memory)

Die `RAM`-Klasse simuliert den Arbeitsspeicher des Computers. Ihrem Konstruktor wird die Grösse des Arbeitsspeichers als Parameter gegeben und damit wird die dementsprechende Anzahl an Bytes, jeweils ein String mit acht Nullen, in eine Liste geschrieben und als Attribut der Klasse gespeichert. Es gibt vier simple Methoden, die zum Arbeitsspeicher gehören.

Die erste Methode `reset(self)` setzt das Objekt zurück in den Anfangszustand. Es werden also alle Bytes auf null gesetzt.

Die zweite und dritte Methode wird benutzt, um in den Speicher zu schreiben und vom Speicher zu lesen. Die Methode `read(self, address)` gibt das Listenelement bei Index `address` zurück. Die Methode `write(self, address, data)` setzt das Listenelement bei Index `address` zu `data`.

Die vierte Methode `load_program(self, program_path)` lädt ein Programm in den Speicher. Die Methode nimmt einen String mit dem Pfad zu einer Datei als Argument. Mit der `open()`-Funktion wird der Inhalt der Datei Zeile für Zeile gelesen und als Liste gespeichert. Mit einer `for`-Schleife wird jedes Element dieser Liste nacheinander in den Arbeitsspeicher geschrieben.

3.4.5 Die Klasse Clock

Die `Clock`-Klasse simuliert den Taktgeber und hat zwei wichtige Methoden.

Die erste Methode ist `tick(self)`. Sie überprüft zuerst, ob die CPU im tick-modus ist. Falls dies der Fall ist, wird zwischen jeder Operation des Von-Neumann-Zyklus, also Befehl laden, Befehl dekodieren und Befehl ausführen, auf einen Knopfdruck im GUI gewartet. Das passiert mithilfe der Threading-Bibliothek. Nach dem Knopfdruck wird die entsprechende Methode der CU ausgeführt und anschliessend wird das GUI aktualisiert.

Die zweite Methode ist `run(self)`. Diese Methode startet eine Schleife die jeweils `tick(self)` aufruft. Dabei wird immer wieder überprüft, ob das Attribut `self.running` der CPU noch `True` ist. Andernfalls wird die Schleife verlassen.

3.4.6 Die Klasse Computer

Die `Computer`-Klasse ist relativ klein. Sie speichert eine Instanz der `CPU`-, `RAM`- und `Clock`-Klasse. Zudem hat sie eine Methode, um den Computer zu starten, und eine Methode, um den Computer herunterzufahren, bei der die `CPU`, die `ALU` und der `RAM` zurückgesetzt werden.

3.4.7 Die Klasse PixelDisplay

Die `PixelDisplay`-Klasse erstellt einen Screen, übersetzt jede Arbeitsspeicher-Adresse in eine Farbe und malt alle Pixel auf den simulierten Monitor.

Die `color_picker(string)` Methode ist eine statische Methode und nimmt einen String als Argument. Basierend auf diesem String wird dann ein Tupel mit dem entsprechenden RGB-Wert zurückgegeben. Insgesamt gibt es 16 verschiedene Farben, die zurückgegeben werden können von den binären Zahlen 0 bis 1111.

Die `translate_ram_to_colors(self)` Methode liest die letzten zwei Kibibyte im Arbeitsspeicher und übersetzt diese mithilfe der `color_picker(string)` Methode in die entsprechenden Farben.

Die `run(self)` Methode erstellt das Pygame-Fenster und enthält eine Schleife, in der immer wieder die `translate_ram_to_colors(self)` Funktion aufgerufen wird, um die RGB-Werte für jeden Pixel zu aktualisieren. Anschliessend werden diese RGB-Werte benutzt, um allen Pixeln die entsprechende Farbe zu geben.

3.4.8 Die Klasse GUI (Graphic User Interface)

Die `GUI`-Klasse ist für die grafische Benutzeroberfläche zuständig. Für das Design habe ich die Vorlage «*Azure theme for ttk*» [12] von *rdhende* benutzt. Die Klasse hat eine Methode `start(self)`, die das `root`-Fenster öffnet. Von diesem Fenster kann man per Knopfdruck die anderen Methoden der `GUI`-Klasse aufrufen, um andere Fenster zu öffnen.

3.4.9 Die Klasse AppController

Die `AppController`-Klasse fungiert als eine sogenannte Vermittlerklasse. Die `GUI`-, `PixelDisplay`- und `Computer`-Klasse kommunizieren über die `AppController`-Klasse miteinander. Bei Applikationen mit mehreren separaten Teilen, ist es oft praktischer, die verschiedenen Teile nicht direkt miteinander kommunizieren zu lassen. Beim Hinzufügen eines neuen Teils kann man das

neue Element einfach mit der Vermittlerklasse verknüpfen, anstatt es mit jedem einzelnen Teil der Applikation zu verbinden.

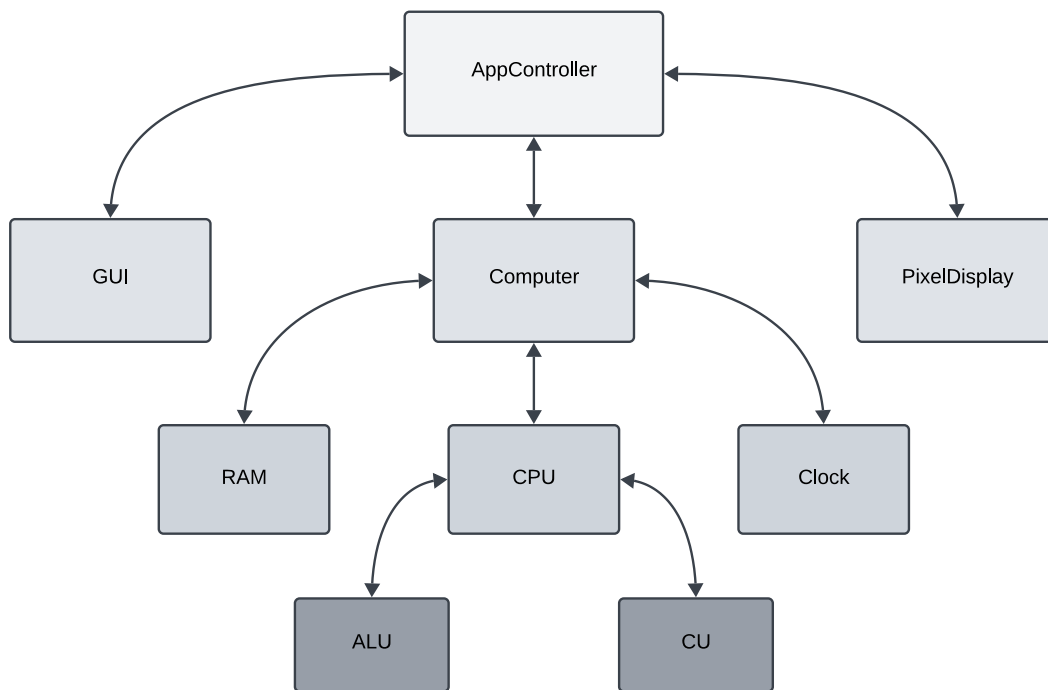


Abbildung 4: Darstellung der Klassenverhältnisse

3.5 Programmlogik

In der `main.py` Datei wird die `main()` Funktion ausgeführt die zwei Dinge tut. Sie setzt die `controller`-Variable auf den Rückgabewert der `initialize_system(ram_size)` Funktion und startet das GUI mit der `controller.gui.start(self)` Methode.

Die `initialize_system(ram_size)` Funktion erstellt eine Instanz jeder Klasse, sodass alle Klassen über den Controller miteinander kommunizieren können. Am Schluss wird der Controller mit einer Instanz des Computers, des GUIs und des Screens zurückgegeben.

Die `controller.gui.start(self)` Methode startet das GUI. Von dort aus kann der Benutzer den Computer und den Screen starten.

3.6 Rolle des Assemblers

Für mich war schon von Anfang an klar, dass ich so etwas wie einen Assembler programmieren werde. Eines meiner Ziele war es, dass der Computer programmierbar sein muss. Der Computer wäre eigentlich auch ohne Assembler programmierbar. Zu Beginn musste ich alle Programme, um die CPU zu testen, mit Einsen und Nullen programmieren. Für sehr einfache Aufgaben, wie das Addieren zweier Zahlen, ist das nicht besonders schwierig. Bei schon leicht komplexeren Aufgaben wird das aber sehr schnell anstrengend und führt häufig zu Fehlern. Alle Zahlen, für Dinge wie Speicheradressen, Farbcodes für die Pixel, Berechnungen und so weiter, müssen in ihrer Binärrepräsentation in das Programm geschrieben werden. Auch die Instruktionen müssen alle codiert werden, denn der Computer versteht nur Einsen und Nullen. Wenn man ein ganzes Programm so schreibt, verliert man schnell die Übersicht. Der Assembler automatisiert diese Prozesse und senkt dabei den benötigten Aufwand. Zwar gibt es keine for-Schleifen und if-else-Anweisungen, aber der Kontrollfluss lässt sich mithilfe von bedingten Sprüngen (Conditional Jumps) steuern, und die Assembler-Sprache hilft dabei, den Überblick zu behalten.

Eine weitere Funktion des Assemblers ist die Benutzung von Labels. Wenn man ohne Assembler programmiert und zum Beispiel eine Jump-Instruktion verwendet, dann muss man die Speicheradresse, zu der man springen will, im Kopf behalten und sie in ihrer Binärrepräsentation hinschreiben. Der Assembler lässt den Programmierer Labels setzen. Er kann an eine bestimmte Stelle im Programm ein Label setzen und es benennen. Wenn er später im Programm zurück an diese Stelle springen will, dann muss er nicht selbst herausfinden, um welche Speicheradresse es sich handelt. Der Assembler löst das Label in die Adresse auf und die Jump-Instruktion setzt den Programm Counter, ohne all den Aufwand, an die gewünschte Stelle. Das ist besonders praktisch, denn die Speicheradresse, zu der man springen will, bleibt während dem Programmieren des Programms nicht immer gleich. Hat man die Adresse, zu der man springen will, einmal herausgefunden und aufgeschrieben, muss man vorsichtig sein, denn wenn man vor dieser Stelle eine Instruktion einfügt oder löscht, dann verschiebt sich auch die Adresse, zu der man springen will. Wenn man Labels benutzt, muss man sich darüber jedoch keine Sorgen machen. Die Labels werden nämlich erst nachdem man das Programm fertig geschrieben hat, vom Assembler aufgelöst. Besonders bei der Benutzung von Labels kommt es manchmal noch zu Bugs im Assembler. Diese Bugs können aber häufig mit dem Löschen von Leerzeichen und leeren Zeilen in der Assembly-Datei behoben werden.

4 Prozessdokumentation

4.1 Entwicklungsprozess

Am Anfang der Maturaarbeit habe ich einfach mal drauflos programmiert. Ich habe mit einem GUI begonnen und anschliessend mit dem Programmieren des Emulators weitergemacht.

Vor der Maturaarbeit-Intensivwoche habe ich noch eine Art Assembler geschrieben, damit ich meine CPU nicht mehr ganz von Hand programmieren muss, indem ich Einsen und Nullen in den Arbeitsspeicher schreibe.

Während der Maturaarbeit-Intensivwoche habe ich Probleme beim Debuggen, Ändern, Löschen und Hinzufügen von Features in meinem Programm gehabt. Ich habe realisiert, dass mein Projekt nicht genügend durchdacht und geplant war. Ich entschied mich dazu, das Programm nochmals von vorne zu schreiben.

Bei meinem zweiten Versuch habe ich mir vor dem Programmieren mehr Gedanken gemacht, wie ich das Programm strukturieren will. Ich habe meine ISA designed und mir überlegt welche Klassen ich brauche. Bei der Planung habe ich realisiert, dass ich einen modulareren Ansatz benötige. Ich muss die einzelnen Teile meiner Applikation isoliert testen und weiterentwickeln können.

Beim Planen habe ich mir überlegt welche Instruktionen ich brauche und wie eine Instruktion aussehen soll. Ich habe mich dazu entschieden alle Instruktionen 32 Bit beziehungsweise 4 Byte lang zu machen. Die ersten 8 Bit stehen für den Opcode, also welche Instruktion ausgeführt wird. Als nächstes gibt es zwei 4-Bit-Operanden die hauptsächlich für Register gebraucht werden, aber auch kleine Sofortoperanden sein können. Die letzten 16 Bit bilden einen Operanden, der hauptsächlich für Sofortoperanden steht. Einen 16-Bit-Operanden zu haben, ist in diesem Fall von grossem Vorteil, da der Arbeitsspeicher einen Adressraum von 2^{16} (65'536 Adressen) besitzt. Mit einem 16-Bit-Operandenfeld kann man direkt auf den gesamten Adressraum zugreifen, was im Vergleich zu mehreren kleinen Operandenfeldern deutlich einfacher und effizienter ist. Für Instruktionen, die einen oder mehrere Operanden nicht benötigen, wird die Stelle einfach mit Nullen aufgefüllt.

Nach dem Planen habe ich mich an die Implementation gemacht und die Instruktionen programmiert sowie die Klassen für die Computerkomponenten ge-

schrieben. Für die neue ISA und das neue Computer-Modell habe ich anschließend wieder einen Assembler programmiert. Der neue Assembler machte es viel einfacher Testprogramme zu schreiben.

4.2 Herausforderungen und Lösungen

Das erste grössere Problem habe ich bereits genannt. Mein erster Ansatz war nicht modular und erlaubte es mir nicht meine Teile effizient einzeln zu testen und erschwerte das Weiterentwickeln des Programms. Als ich im Internet gesucht habe, wie man objektorientierte Projekte übersichtlicher macht, bin ich auf das Konzept einer Vermittlerklasse [13] gestossen. Die einzelnen Teile des Projektes wie zum Beispiel das GUI, der Monitor und der Computer sollen nicht direkt miteinander kommunizieren. Die verschiedenen Komponenten sollten nur über die Vermittlerklasse miteinander kommunizieren.

Die zweite Herausforderung war der Bildschirm. Die Idee hinter dem Bildschirm ist relativ simpel. Der Inhalt des RAMs soll grafisch dargestellt werden und repetitiv aktualisiert werden. Dafür muss lediglich jedem der 4096 Pixel bei jeder Aktualisierung die korrekte Farbe zugewiesen werden. Um irgendetwas überhaupt grafisch darstellen zu können, brauchte ich die passende Bibliothek. Zuerst versuchte ich das Display mit dem Canvas Widget von tkinter zu programmieren. Es hat zwar funktioniert, wurde aber bei längerer Laufzeit immer langsamer und ich konnte nicht herausfinden, woran es lag. Ich musste also eine Alternative finden und glücklicherweise eignete sich Pygame dafür.

Da der Computer, der Bildschirm und das GUI parallel laufen müssen, brauchte ich das Threading-Modul. Ich habe versucht das GUI im Hauptthread laufen zu lassen und den Bildschirm, für den ich zu dieser Zeit noch tkinter's Canvas benutzt habe, in einem Nebenthread auszuführen. Tkinter ist allerdings nicht thread-safe, was bedeutet, dass man das tkinter-Modul nicht in einem Nebenthread benutzen kann. Da ich aber zwei Prozesse hatte, die beide tkinter benutzten und deshalb nicht im Nebenthread laufen dürfen, war es nicht möglich sie parallel auszuführen. Als ich für den Bildschirm Pygame verwendet habe, konnte ich das GUI im Hauptthread laufen lassen und den Computer sowie den Bildschirm in einem Nebenthread ausführen, denn Pygame ist thread-safe.

Eine andere Herausforderung bereitete mir der Umgang mit Bibliotheken und Imports. Am Anfang war es schwierig den Überblick zu behalten. Auch das korrekte Initialisieren der verschiedenen Objekte war nicht einfach. Um alles ordentlich zu halten, erstellte ich die `init.py` Datei, in der alle Initialisierungen stattfinden.

5 Analyse und Bewertung

5.1 Bewertung des Produkts

Der emulierte Computer basiert auf einer selbst entworfenen ISA und kann mithilfe einer mit tkinter implementierten Benutzeroberfläche bedient werden. Gleichzeitig zeigt diese Benutzeroberfläche dem Benutzer an, was genau im Moment im Computer vor sich geht und erlaubt es ihm Taktzyklus für Taktzyklus durch das Programm zu gehen, um wirklich zu verstehen, was passiert. Es zeigt zu jeder Zeit an, an welchem Punkt im Von-Neumann-Zyklus sich der Computer befindet, welche Instruktion gerade im Befehlsregister ist, und zeigt auch die Inhalte der anderen Register an.

Das Programm könnte beispielsweise im Unterricht verwendet werden. Die Lehrperson kann mithilfe des Assemblers eigene Programme schreiben und den Schülern zeigen, was all die Komponenten im Computer machen. Die Schüler können anschliessend andere Programme allein testen. Sie haben die Möglichkeit das Programm automatisch laufen zu lassen, um zu sehen, was das Programm macht. Danach können sie Taktzyklus für Taktzyklus durch das Programm gehen und versuchen zu verstehen, was genau Schritt für Schritt gemacht wird.

5.2 Analyse

Um den Emulator zu testen, habe ich ein einfaches Programm geschrieben, das alle Pixel nacheinander von schwarz auf grau setzt. Dafür wird zunächst die binäre Zahl für die Farbe sowie die Adressen im Arbeitsspeicher für den Pixel, der als nächstes auf hellblaugesetzt werden soll, und den letzten Pixel jeweils in ein separates Register geladen. An diesem Punkt wird das Label `loop` für die Schleife gesetzt, damit die `jne` (*jump not equal*) Instruktion zurück zu diesem Ort springen kann. Die `store` Instruktion speichert die Farbe an der Adresse, die im Register für den nächsten Pixel steht. Die Adresse wird um 2 erhöht und zeigt somit auf den nächsten Pixel. Die `cmp` (*compare*) Instruktion schaut, ob die Adresse für den nächsten Pixel gleich ist, wie die Adresse für den letzten Pixel. Falls dies nicht der Fall ist, springt die `jne` Instruktion zurück zum Label `loop`. Andernfalls springt die `jne` Instruktion nicht zurück und die `halt` Instruktion, die das Programm beendet, wird ausgeführt.

Nachdem ich das Programm in der Assemblysprache geschrieben habe, habe ich sie mithilfe des Assemblers in Maschinencode übersetzt. Mit der `main.py`

Datei habe ich den Emulator gestartet. Auf der grafischen Benutzeroberfläche konnte ich mit den Knöpfen alle anderen Fenster öffnen, um zu sehen, welche Prozesse im Computer ablaufen. Zuerst bin ich Taktzyklus für Taktzyklus durch das Programm. So konnte ich sehen, welche Instruktionen ausgeführt wurden und was in den Registern gespeichert war. Nach einer Weile erschienen die ersten Pixel auf dem Bildschirm und das Programm funktionierte.

Danach habe ich getestet, wie lange es geht, bis alle Pixel grau wurden, wenn ich das Programm automatisch laufen gelassen habe. Das Ausführen des Programms dauerte 5 Minuten und 3 Sekunden. Insgesamt wurden 4'101 Instruktionen ausgeführt. Die Taktfrequenz beträgt demnach ungefähr 13.52 Hz. Ein Intel® Core™ i5 Prozessor [14] hat eine Taktfrequenz von ungefähr 4.7 GHz. Das entspricht fast 350 Millionen-mal mehr Instruktionen pro Sekunde. Obwohl ich mich nicht primär auf die Leistung des Prozessors fokussiert habe, habe ich noch ein bisschen Zeit in die Optimierung investiert. Ich habe einige `print`-Befehle entfernt und das Programm lief schon viel schneller. Mit 3'000 Instruktionen pro Sekunde war ich aber noch nicht ganz zufrieden. Nach weiterem Optimieren der grafischen Benutzeroberfläche, des Prozessors und des gesamten Emulators im Generellen, konnte ich die Anzahl Instruktionen pro Sekunde auf 60'000 erhöhen. Je nach Programm und ob der Bildschirm geöffnet ist, kann diese Zahl schwanken.

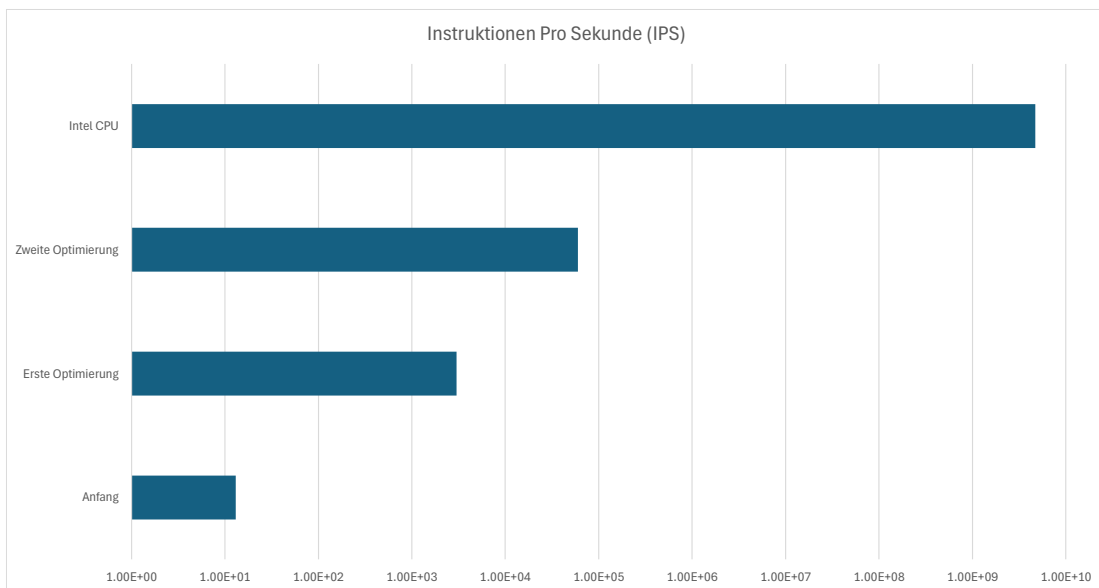


Abbildung 5: Instruktionen pro Sekunde (Logarithmisches Balkendiagramm)

Da die CPU Turing-vollständig ist, kann sie in der Theorie jedes berechenbare Problem lösen, vorausgesetzt sie hat genug Zeit und Speicherplatz. Mit dem Emulator könnte man theoretisch komplexe wissenschaftliche Berechnungen wie beispielsweise ein Klimamodell berechnen. Komplexe wissenschaftliche Berechnungen bestehen aus einer Vielzahl von mathematischen Operationen, von Differenzialgleichungen bis Matrixmultiplikationen. Da diese Berechnungen auf den Regeln der Mathematik und Logik basieren, kann sie in der Theorie jede Turing-vollständige Maschine, also auch die CPU, ausführen.

In der Praxis hat mein Emulator aber einige Faktoren, die das Ausführen von komplexeren Programmen einschränkt. Programme mit hohem Rechenaufwand können auf meinem Emulator kaum ausgeführt werden. Ein Programm, das auf einem modernen Prozessor in einer Stunde ausgeführt werden kann, würde auf meiner CPU vielleicht Jahre dauern. Ein weiteres Problem ist die Genauigkeit. Viele Programme arbeiten mit Gleitkommazahlen und fordern teilweise hohe Präzision. Da mein Prozessor keine Gleitkommazahlen unterstützt wären Berechnungen zu ungenau. Eine weitere Einschränkung ist der begrenzte Speicher. Der Arbeitsspeicher meines Emulators umfasst lediglich 64 KiB. Zum Vergleich: Der Windows-Texteditor benötigt etwa 31 MB RAM, was mehr als das 470-fache des verfügbaren Speichers meines Emulators entspricht. Zudem hat mein Emulator keinen optimierten Cache oder schnellen Speicherzugriff implementiert. Das ständige Zugreifen auf den Arbeitsspeicher würde die Geschwindigkeit weiter verlangsamen. Der Emulator ist ausserdem auf einen einzelnen Prozessor beschränkt. Anders als bei Multi-Core-Architekturen fehlt bei meinem Emulator die Möglichkeit Prozesse parallel laufen zu lassen.

Obwohl der Emulator theoretisch jedes berechenbare Problem lösen kann, kann er in der Praxis viele Programme nicht in angemessener Zeit ausführen. Die grössten Einschränkungen sind die Grösse des Arbeitsspeichers und die langsame Ausführung der Instruktionen, die auch nach der Optimierung im Vergleich zu modernen Prozessoren sehr schwach ist.

5.3 Reflexion

Während der Entwicklung meines Produktes habe ich einiges gelernt. Natürlich habe ich schon oft gehört wie wichtig Planung und sauberer Code ist, bei diesem Projekt habe ich aber bemerkt, warum diese Dinge so wichtig sind. Ich bin einerseits froh, dass ich mich direkt ins Projekt gestürzt habe, denn der schwierigste Teil ist oftmals sich dazu zu bringen endlich anzufangen. Andererseits

hätte ich viel Zeit und Nerven gespart, wenn ich von Anfang an besser geplant hätte und mich mehr mit der Theorie auseinandergesetzt hätte.

Eine Sache, die ich im Nachhinein definitiv anders machen würde, ist die Benutzung einer Vermittlerklasse. Vor dem Projekt wusste ich noch nichts davon, wie man seinen Code modular gestaltet. Jetzt weiss ich wie viel übersichtlicher eine grössere Menge an Code sein kann, wenn man seinen Code modular gestaltet.

5.4 Ausblick

Das Produkt lässt sich auf jeden Fall noch weiterentwickeln. Eine Sache, die ich noch implementieren möchte, ist ein Interrupt-Handler. Ohne Interrupt-Handler lässt sich nämlich kein Programm mit Benutzereingabe schreiben. Eine andere Sache, die ich gerne noch implementieren würde, ist ein Stack. Ein Stack würde die Implementierung von Funktionen vereinfachen und somit das Programmieren von Anwendungen für die CPU noch einfacher machen. Abgesehen davon lässt sich der Emulator wahrscheinlich noch weiter optimieren.

6 Fazit

6.1 Zusammenfassung

Der entwickelte CPU-Emulator basiert auf einer selbst entworfenen ISA und kann mit einer grafischen Benutzeroberfläche, die in Python mit tkinter implementiert wurde, bedient werden. Der Assembler ermöglicht es dem Benutzer eigene Programme in Assembly zu schreiben und diese auf dem Emulator auszuführen. Da der Emulator Turing-vollständig ist, lässt sich mit ihm in der Theorie jedes berechenbare Problem lösen.

Die grafische Benutzeroberfläche lässt den Benutzer den emulierten Computer bedienen und hilft mit den Visualisierungen der internen Prozesse dem Benutzer die Funktionsweise eines Computers zu verstehen.

Obwohl der Emulator theoretisch jedes berechenbare Problem lösen kann, hat er in der Praxis mit komplexen und modernen Anwendungen Schwierigkeiten. Das liegt hauptsächlich an der geringen Leistung und dem begrenzten Arbeitsspeicher von 64 KiB. Ohne Gleitkommazahlen fehlt dem Emulator bei Programmen, die Genauigkeit erfordern, die Präzision.

In Zukunft plane ich die Funktion des Emulators mit einem Stack und einem Interrupt-Handler zu erweitern sowie die Leistung der CPU zu erhöhen.

6.2 Abschliessende Gedanken

Die Entwicklung des Emulators war für mich eine spannende und lehrreiche Herausforderung. Ich habe nicht nur mein Wissen über Computerarchitektur vertieft, sondern auch viel über den Wert einer gründlichen Planung und eines modularen Designs gelernt. Auf die selbst entwickelte ISA und den Assembler, mit denen man einfache Programme für den Emulator schreiben und ausführen kann, bin ich besonders stolz.

Insgesamt hat mir das Projekt gezeigt, wie vielschichtig und herausfordernd die Entwicklung von Computersystemen ist.

Verweise

- [1] «YouTube,» [Online]. Available:
<https://www.youtube.com/watch?v=osFa7nwHHz4&t=142s>. [Zugriff am März 2024].
- [2] «YouTube,» [Online]. Available:
<https://www.youtube.com/watch?v=5rg7xvTJ8SU&t=235s>. [Zugriff am März 2024].
- [3] «Informatik Mygymer,» [Online]. Available:
<https://informatik.mygymer.ch/base/?book=computer>. [Zugriff am Oktober 2024].
- [4] «YouTube,» [Online]. Available:
<https://www.youtube.com/watch?v=RPQD7-AOjMI>. [Zugriff am September 2024].
- [5] «Python,» [Online]. Available: <https://docs.python.org/3/library/sys.html>. [Zugriff am Oktober 2024].
- [6] «Python,» [Online]. Available:
<https://docs.python.org/3/library/pathlib.html>. [Zugriff am Oktober 2024].
- [7] «Python,» [Online]. Available:
<https://docs.python.org/3/library/threading.html>. [Zugriff am Oktober 2024].
- [8] «Python,» [Online]. Available:
<https://docs.python.org/3/library/tkinter.html>. [Zugriff am Oktober 2024].
- [9] «Python,» [Online]. Available: <https://docs.python.org/3/library/os.html>. [Zugriff am Oktober 2024].
- [10] «Pygame,» [Online]. Available: <https://www.pygame.org/>. [Zugriff am Oktober 2024].
- [11] «Python,» [Online]. Available: <https://docs.python.org/3/library/time.html>. [Zugriff am Oktober 2024].

- [12] «GitHub,» [Online]. Available: <https://github.com/rdbende/Azure-ttk-theme>. [Zugriff am August 2024].
- [13] «Refactoring Guru,» [Online]. Available: <https://refactoring.guru/design-patterns/mediator>. [Zugriff am August 2024].
- [14] «Intel,» [Online]. Available: <https://www.intel.de/content/www/de/de/products/details/processors/core/i5/products.html>. [Zugriff am 10 2024].

Abbildungsverzeichnis

Abbildung 1: Von-Neumann-Architektur [3].....	8
Abbildung 2: Von-Neumann-Zyklus [3].....	9
Abbildung 3: Aufbau der CPU	15
Abbildung 4: Darstellung der Klassenverhältnisse	18
Abbildung 5: Instruktionen pro Sekunde (Logarithmisches Balkendiagramm) ..	23

Programmier- und Schreibhilfen

- <https://code.visualstudio.com/> VSC
- <https://www.python.org/> Python
- <https://github.com/features/copilot> GitHub Copilot
(VSC-Plug-in für automatische Vervollständigung, nicht zur Generierung von ganzen Code-Blöcken)
- <https://chatgpt.com/> ChatGPT
(Korrektur der Rechtschreibung und Grammatik von einzelnen Sätzen oder kleinen Teilen, nicht zur Generierung von Paragrafen oder Textabschnitten)