OST
Ostschweizer
Fachhochschule

Prozessor-Simulator

Bachelorarbeit

Studiengang Informatik

OST – Ostschweizer Fachhochschule

Campus Rapperswil-Jona

Frühjahrssemester 2021

Autoren:	Yves Boillat, Eliane Schmidli
Betreuer:	Prof. Stefan Richter
Experte:	Dr. Ettore Ferranti
Gegenleser:	Prof. Beat Stettler

Danksagung:
Wir möchten hier allen Personen danken, die uns bei der Bachelorarbeit unterstützt haben. Vor allem danken wir unserem Bachelorarbeitsbetreuer Prof. Stefan Richter und Florian Bruhin für die vielen wertvollen Inputs und die grosse Unterstützung während des Projekts. Zudem danken wir Hanspeter Schmidli und Irène Schwander für das Korrekturlesen der Arbeit. Während des Projekts führten wir diverse Tests durch und möchten den Testpersonen herzlich danken, die sich Zeit genommen und zur Verbesserung der Qualität des Prozessor-Simulators beigetragen haben.
Dies sind:

Kevin Löffler
Leonard Schuetz
Marc Havrilla
Olivier Lischer
Ramon Ebneter
Saskia Stillhart
Sebastian Lehner
Stefan Schmidli

1 Inhalt

2 Zusammenfassung

2.1 Aufgabenstellung

Bachelorarbeit «Prozessorsimulator»

Einführung

Für das Verständnis von Betriebssystemen und hardwarenaher Software ist ein Verständnis der Hardware auf abstraktem Niveau unabdinglich. Dafür soll ein Prozessor-Simulator entwickelt werden, der es Studierenden ermöglicht, die Funktionsweise der Hardware abstrakt nachzuvollziehen.

Es gibt bereits einige Prozessor-Simulatoren, z.B. https://sourceforge.net/projects/johnnysimulator/ oder https://draemm.li/various/mirac/. Leider sind diese aber detaillierter als benötigt und simulieren selten echte CPU-Sätze. Sie sind eher für das Studium des Prozessorbaus interessant denn für hardware-nahe Programmierung.

Die Studierenden haben in ihrer Studienarbeit im HS 20/21 bereits die Grundlagen für ein solches Tool gelegt. Nun soll das Tool in die Produktreife überführt werden, so dass es im Unterricht eingesetzt werden kann.

Aufgabe

Ziel dieser Arbeit soll es sein, aufbauend auf der Studienarbeit einen grafischen Prozessor-Simulator zu erstellen, der folgende Funktionen erfüllt:

- Ausführen der für das Modul Betriebssysteme 1 benötigten x86-64 Prozessor-Instruktionen
- Anzeigen des Zustands aller Register, des Speicherbus und festlegbarer Speicherbereiche
- Ansprechende Darstellung der entsprechenden Zustandsübergänge und des gesamten Prozessorzyklus (konzeptionell)
- Tabellarische Darstellung der historischen Zustände nach jedem durchgeführten Schritt

Der entwickelte Code soll Open-Source sein. Es sollen und dürfen soweit als möglich bestehende Open-Source-Systeme und -Bibliotheken verwendet werden, bspw: https://www.unicorn-engine.org/. Die Wahl der Tools und Programmiersprache obliegt den Studierenden. Die entwickelte Software soll möglichst direkt von den Studierenden einsatzbar sein; Stabilität und Benutzbarkeit der Software sowie deren Erweiterbarkeit sind wichtiger als der Funktionsumfang.

Die Studierenden sollen strukturiert und ingenieurmässig vorgehen und wissenschaftlich nachvollziehbare Untersuchungen durchführen und dokumentieren.

Termine

Die Bachelorarbeit beginnt am 24.2.2021. Abgabetermin ist der 18.6.2021.

Betreuung

Die Bachelorarbeit wird durch Prof. Stefan Richter betreut. Jeden Mittwoch von 15:00 bis 16:00 findet eine Besprechung statt, in der die Studierenden den Fortschritt der Arbeit präsentieren.

Fragen und Angelegenheiten können ausserhalb dieses Termins auch per E-Mail erörtert werden.

Experte für die Bewertung ist Dr. Ettore Ferranti, Product Line Manager for Digital+Services at Hitachi ABB Power Grids.

Bewertung

Die Arbeit wird anhand der folgenden sechs gleichgewichteten Punkte bewertet:

1. Organisation, Durchführung (Projektplanung u. Nachführung Arbeit gemäss Projektplan, Selbständigkeit, Einsatz, Zusammenarbeit mit Auftraggeber, Betreuer)
2. Bericht (Inhalt des Projektschlussberichts, Gliederung, Darstellung, Sprache der gesamten Dokumentation)
3. Problemanalyse (Vorstudie, Literaturstudium, Anforderungsspezifikation, Anforderungsanalyse, Domainanalyse)
4. Lösungsentwurf (Lösungsvarianten und deren Beurteilung, Variantenentscheid, Konzept, Entwurf)
5. Realisierung und Test
6. Präsentation und mündliche Prüfung

Hinweise

Die folgende Seite bietet zahlreiche nützliche Informationen zum Schreiben einer wissenschaftlichen oder technischen Arbeit:

https://www.ifs.hsr.ch/index.php?id=13194&L=4metadata%2Foai_dc_1.dc

2.2 Abstract

Die Arbeitsweise des Prozessors ist Teil des Moduls "Betriebssysteme 1" des Informatik Studiums an der OST - Ostschweizer Fachhochschule. Für das bessere Verständnis entwickelten wir in der Studienarbeit “Grafischer Prozessor-Simulator” einen Prototypen, der nun in die Produktreife überführt werden soll. Zu diesem Zweck setzen wir den in der Studienarbeit entwickelten und verifizierten Designansatz um und ergänzen den Simulator um die Möglichkeit eigenen Code einzugeben. Durch die Portierung und Einbindung des Netwide Assembler (Nasm) in unser Webprojekt ermöglichen wir den Studierenden Code aus der Vorlesung sowohl im Simulator als auch auf dem Betriebssystem auszuführen. Zusätzlich entwickeln wir eine Methode, um die Ausführung von Instruktionen des x64-Befehlssatzes durch die CPU generisch darzustellen. Der Simulator wird von uns um weitere vorlesungsrelevante Inhalte wie der Stack erweitert. Das aus der Arbeit resultierende Produkt zeigt Zustandsänderungen ansprechend und nachvollziehbar an und hebt sich somit von anderen Simulatoren ab. Die Verständlichkeit und Benutzbarkeit des Prozessor-Simulators wurden mit Hilfe von Usability Tests nachgewiesen. Unser Produkt kann zur Erklärung der Arbeitsweise eines Prozessors auf einem für die Informatik Studierenden relevanten Abstraktionsniveau eingesetzt werden. Wir schlagen weitere Funktionalitäten vor, die in einer Folgearbeit ergänzt werden könnten. Der Simulator ist so aufgebaut, dass dieser ohne grösseren Aufwand erweitert werden kann.

2.3 Lay Summary

Der Prozessor ist ein zentraler Bestandteil des Computers der Befehle ausführt. Zum Beispiel liest er Daten aus dem Speicher, rechnet damit und schreibt das Resultat wieder zurück in den Speicher. Im Modul “Betriebssysteme 1” wird den Informatik Studierenden an der OST - Ostschweizer Fachhochschule der Prozessor und seine Arbeitsweise nähergebracht.

Um den Studierenden beim Verständnis dieses Themas zu helfen, entwickelten wir in unserer Studienarbeit “Grafischer Prozessor-Simulator” im Herbst 2020 ein Konzept für einen Simulator. Er soll den Studierenden visuell zeigen, wie der Prozessor schrittweise Befehle eines Programms lädt, verarbeitet und ausführt. Das Ziel der Bachelorarbeit ist nun diesen Simulator fertigzustellen, damit er im Unterricht eingesetzt werden kann.

Um das Ziel zu erreichen, setzen wir das erarbeitete Konzept um und ergänzen den Simulator mit der Möglichkeit, dass die Studierenden Programme, die sie in der Vorlesung kennenlernen, eingeben können. Mithilfe von Animationen veranschaulicht der Simulator was der Prozessor mit dem Programmcode macht. Wir testen die Verständlichkeit des Simulators mit Studierenden und überprüfen, ob es Schwierigkeiten mit der Bedienung gibt.

Das aus der Arbeit resultierende Produkt begeistert die Studierenden und kann somit im Unterricht eingesetzt werden. Im Gegensatz zu bereits bestehenden Simulatoren ist unser Produkt verständlicher und wirkt ansprechender. Es eignet sich für Personen mit unterschiedlichsten Vorkenntnissen und hilft ihnen ein tiefes Verständnis des Vorlesungsstoffs zu erhalten. Bei der Umsetzung haben wir darauf geachtet, dass die Möglichkeit besteht in einer Folgearbeit noch weitere Funktionalitäten hinzuzufügen.

3 Ausgangslage

Diese Arbeit ist eine Folgearbeit zu unserer Studienarbeit “Grafischer Prozessor-Simulator”, die wir im Herbstsemester 2020 durchgeführt haben [1]. Das Resultat der Arbeit besteht aus einem Design-Ansatz für das Graphical User Interface (GUI) (im Microsoft PowerPoint erstellt) und einem programmierten Prototypen. Die Arbeit und die daraus resultierenden Produkte sind in den folgenden Abschnitten zusammengefasst.

3.1 Anforderungen Studienarbeit

Wir erhielten damals die Aufgabenstellung, einen grafischen Prozessor-Simulator zu entwickeln, der zur Vorlesung “Betriebssysteme 1” (Bsys1) passte. Die bestehenden Simulatoren waren zu detailliert für diese Vorlesung und eher für die Elektrotechnik geeignet. Die Anforderungen waren, dass der Simulator ein Subset an x64 Prozessor-Instruktionen schrittweise ausführen sollte und dabei den Zustand aller Register, den Speicherbus und festlegbare Speicherbereiche sinnvoll darstellte. Ausserdem sollten Zustandsänderungen ansprechend animiert werden. Der Code des Projekts sollte Open Source sein.

3.2 Konzept Studienarbeit

Beim Untersuchen bestehender Simulatoren zu Beginn der Arbeit bemerkten wir, dass jeweils viel Zeit in die Programmierung der Prozessor Emulation investiert wurde und die Darstellung des GUI eher zweitrangig war. Wir entschieden uns deshalb, die Unicorn Engine[1] als Emulator zu verwenden und den Fokus des Projekts auf eine verständliche Darstellung des Vorlesungsstoffs im GUI zu legen. Um eine einfache Installation zu gewährleisten und eine für alle Betriebssysteme kompatible Lösung anzubieten, entschieden wir uns, den Simulator als Webprojekt mit dem Framework Vue.js[2] zu entwickeln. Der Simulator soll als HTML-File, das die Studierenden unabhängig von ihrem Betriebssystem im Chrome-Browser öffnen und ausführen können, ausgeliefert werden.

Im GUI wollten wir, dass die Studierenden den Prozessor-Zyklus erkennen können. Es sollte also ersichtlich sein, dass die Central Processing Unit (CPU) eine Instruktion liest, diese ausführt und danach den Instruction-Pointer erhöht, um wiederum die nächste Instruktion zu

[1] Unicorn: https://www.unicorn-engine.org/
[2] Vue.js: https://vuejs.org

lesen. Um ein modernes Design zu verwenden, haben wir uns entschieden Quasar[3] (eine GUI Komponenten Library) zu verwenden, die die Guidelines von Google Material Design umsetzt.

3.3 GUI Designansatz

Die Schwierigkeit beim GUI lag im Finden einer geeigneten Abstraktion, die der Vorlesung gerecht wird. Um die Verständlichkeit unseres GUI zu gewährleisten, entwickelten wir einen animierten Design-Ansatz im Microsoft PowerPoint (siehe Abbildung 1), den wir mit Studierenden des Moduls Bsys1 auf das Verständnis testeten.

Der Ansatz beinhaltete eine Codeeingabe, die ein einfaches Programm zeigte. Danach wurden die Komponenten des Simulators einzeln eingeblendet, um die Informationsüberlastung zu reduzieren. Es wurde eine CPU-Komponente mit den Informationen über die aktuelle Instruktion, den Registern und Flags dargestellt. Zudem gab es eine Memory-Komponente, die einen Bereich mit Instruktionen zeigte, die als Maschinencode dargestellt wurden und einen zweiten mit Speicherdaten, ebenfalls als Bytes dargestellt.

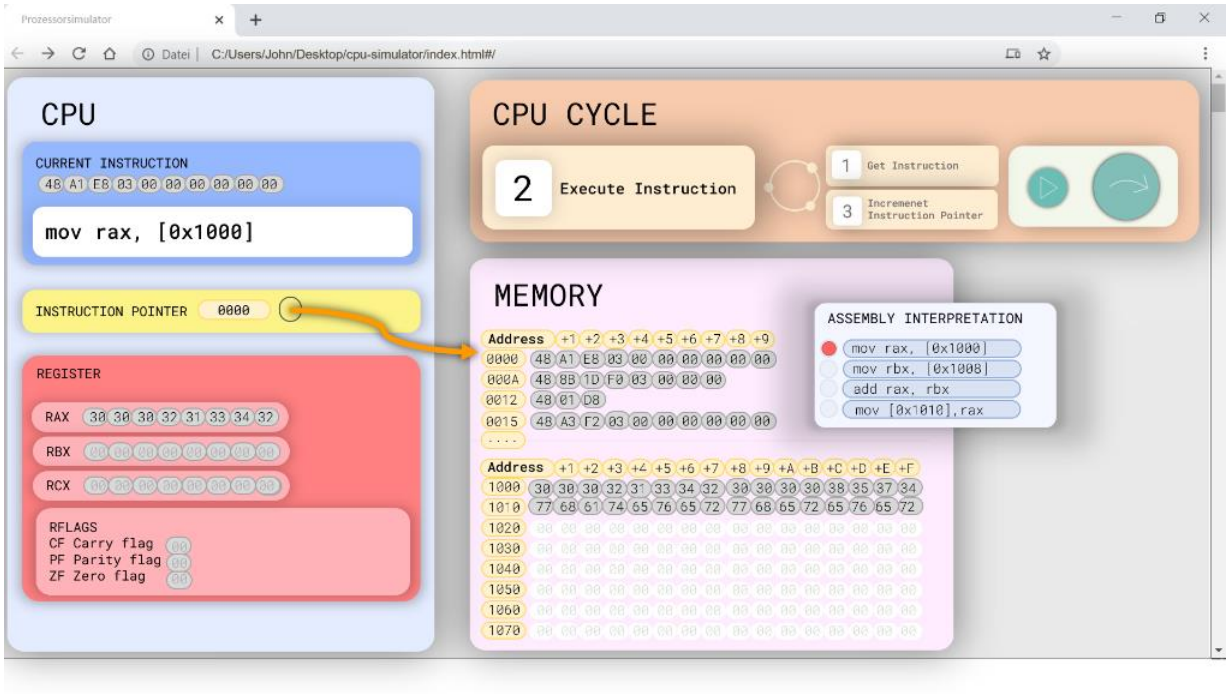


Abbildung 1 Designansatz Studienarbeit

Mithilfe eines Buttons konnte man jeweils den nächsten Schritt des Prozessor-Zyklus starten. Anhand von Pop-ups wurden den Benutzenden Informationen dazu eingeblendet, was

[3] Quasar Framework: https://quasar.dev

in diesem Schritt passierte. Im Schritt “Get Instruction” wurde die Instruktion aus dem Memory gelesen und in die CPU transportiert, wo sie in Assembly übersetzt wurde. Im Schritt “Execute Instruction” wurde in der PowerPoint Präsentation eine “mov” Instruktion ausgeführt vom Speicher zum Register RAX. Die Bytes bewegten sich dabei entlang von Pfeilen zur Darstellung der aktuellen Instruktion in der CPU und danach zum Register. Im Schritt “Increment Instruction Pointer” bewegten sich die Maschinen Bytes aus der aktuellen Instruktion zu einem Zähler beim Instruction Pointer. Der Endstand des Zählers wurde dann zur Adresse im Instruction Pointer hinzuaddiert und der Pointer, der als Pfeil dargestellt wurde, zeigte auf die nächste Instruktion.

Während den Animationen wurden Elemente, die nicht verwendet wurden, leicht ausgeblendet. Ausserdem pulsierten die betroffenen Elemente kurz, bevor sie sich bewegten, um den Animationen besser folgen zu können. Um die Informationsüberlastung zu reduzieren, wurden alle nicht verwendeten Bytes heller dargestellt.

3.4 Prototyp

Nachdem wir den Designansatz getestet und die Verständlichkeit sichergestellt hatten, realisierten wir die wichtigsten Komponenten und Funktionalitäten des Design-Ansatzes in einem Prototypen (siehe Abbildung 3). Dieser beinhaltete die Unicorn Engine, die die Daten des aktuellen Zustands zur Verfügung stellte. Diese wurden in ein **State** Objekt (siehe Abbildung 2) abgefüllt und an das GUI übergeben. Ausserdem wurde der Capstone[4] Disassembler verwendet, um den Maschinencode für die Darstellung der aktuellen Instruktion in Assembly Code umzuwandeln. Dabei wurde das ganze Programm zu Beginn als Maschinencode übergeben und die einzelnen Instruktionen mit der dazugehörigen Assembly Interpretation und der Adresse im Memory gespeichert. Anhand des Instruction Pointers wurde dann die aktuelle Instruktion gesucht. Da die Codeeingabe noch nicht implementiert wurde, musste man den Maschinencode direkt im Sourcecode anpassen.

[4] Capstone: https://www.capstone-engine.org/

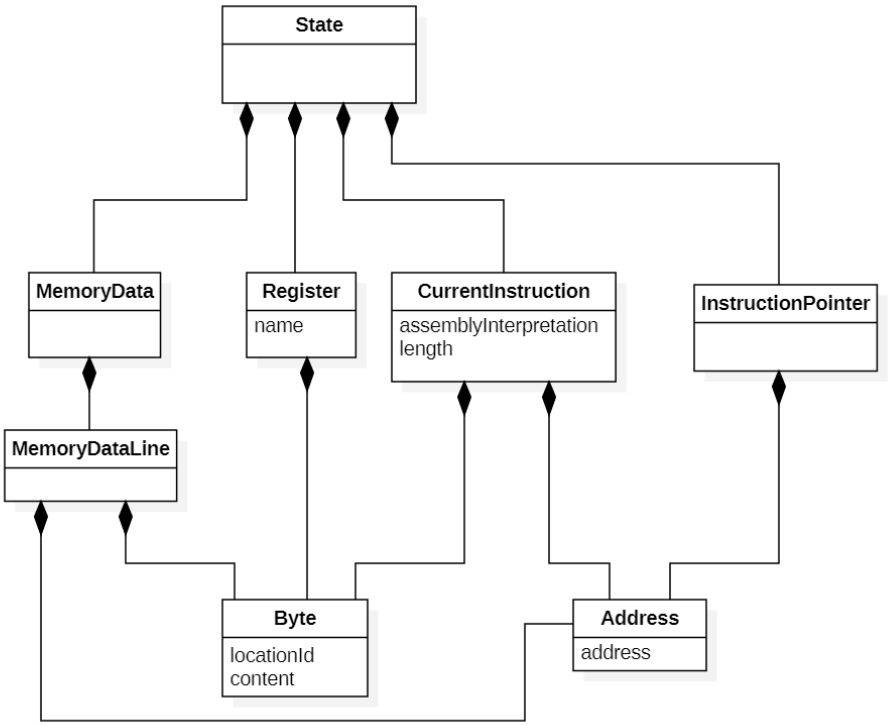


Abbildung 2 Das State Objekt der Studienarbeit

Von den Prozessor-Zyklus Schritten animierten wir bereits “Get Instruction” und eine vereinfachte Version von “Increment Instruction Pointer”. Bei “Execute Instruction” programmierten wir die Animationslogik für die “mov” Instruktion, die Bytes von einem HTML-Element zum anderen bewegt. Dabei werden die Start- und Ziel-Elemente der Animation im HTML-Dokument anhand der einzigartigen HTML-Document-Id gesucht. Bei den Bytes entspricht diese zum Beispiel dem Attribut **LocationId**.

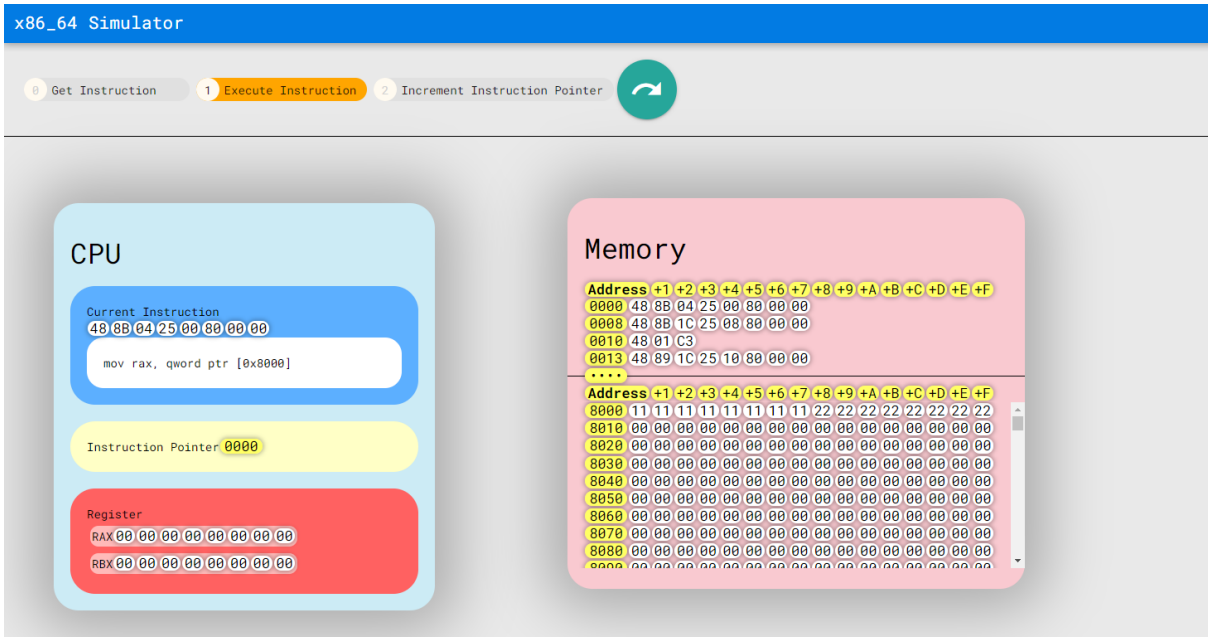


Abbildung 3 Prototyp der Studienarbeit

Da die Instruktionserkennung noch fehlte, hinterlegten wir gewisse Kombinationen von Operanden als hartkodierte Strings. Bei allen anderen Instruktionen wurde der Schritt “Execute Instruction” nicht animiert, sondern lediglich die Daten aktualisiert. Die Idee war, dass in einer Folgearbeit für weitere Instruktionen spezifische Animationen konzipiert und hinzugefügt werden.

3.5 Ergebnis Studienarbeit

Der GUI-Designansatz erfüllte unsere Ziele und war für die Studierenden verständlich. Viele von ihnen gaben an, dass sie sich einen solchen Simulator für die Vorlesung wünschten. Im Prototyp wurde aus Zeitgründen der Code Editor, die Beschreibungstexte, sämtliche Pfeile und die Flags weggelassen. Performance Tests ergaben, dass unser Konzept für die Animation nicht optimal war. In unserem Konzept haben wir den “Get Instruction” Schritt vereinfacht und ausgelassen, dass der Prozessor die Bytes einzeln einliest. Ausserdem sollte die Steuerung noch einmal überdacht werden.

4 Anforderungen

Es gelten immer noch dieselben Anforderungen, wie in der Studienarbeit (siehe Kapitel Anforderungen Studienarbeit). Zum Beispiel sollen der gesamte Prozessor-Zyklus und die Zustandsübergänge ansprechend dargestellt werden. Zudem soll der entwickelte Prototyp in dieser Arbeit in die Produktreife überführt werden, damit er im Unterricht eingesetzt werden kann. Dazu kommt die neue Anforderung an die Funktionalität, die historischen Zustände nach jedem durchgeführten Schritt tabellarisch darzustellen.

4.1 Personas

Bereits in der Studienarbeit haben wir einen typischen Beispielbenutzer des Simulators namens Andreas definiert. Dieser möchte in der Vorlesung mehr über hardwarenahe Programmierung erfahren [1]. Zu Beginn der Bachelorarbeit kategorisierten wir noch einmal die Studierenden, die in der Studienarbeit am Usability Test teilgenommen hatten und erkannten, dass wir dieses Profil in zwei Typen aufteilen können. Es gibt Andreas, der Mühe hat der Vorlesung zu folgen, und wir haben neu auch Charlotte, die ihren Mitstudierenden weit voraus ist.

4.1.1 Andreas (21):

Andreas hat nach seiner Informatiklehre das Informatikstudium an der OST begonnen. Zusammen mit seinen Freunden besucht er das Grundlagenmodul Bsys1. Er möchte erfahren, wie der Prozessor seine Programme ausführt. In der Vorlesung der zweiten Woche ist es so weit und er wird in das Prozessormodell eingeführt. Er hat jedoch Mühe der Vorlesung zu folgen, da es ihm noch schwerfällt, das neu Gelernte richtig einzuordnen.

Ziel im Prozessor-Simulator:
Andreas ist froh, wenn er nicht mit Informationen überladen wird, damit er schnell einen Überblick über die Komponenten erhält. Er möchte Ausführungen der Code Beispiele aus der Bsys1 Vorlesung sehen, um den vermittelten Stoff besser zu verstehen. Ausserdem möchte er in seinem Tempo vorwärts gehen, da er noch etwas Mühe hat, die einzelnen Schritte nachzuvollziehen.

4.1.2 Charlotte (22):

Nach ihrer Lehre als Elektronikerin hat sich Charlotte für ein Informatik Studium an der OST entschieden. Ihr Lieblingsmodul ist Bsys1, da sie dort schon viele Vorkenntnisse hat. Dementsprechend ist sie ihren Mitstudierenden weit voraus. Ihr Ziel ist es, ihre Kenntnisse in hardwarenaher Programmierung zu vertiefen. Deshalb tüftelt sie gerne in den Übungen an ihrem Assembly Code.

Ziel im Prozessor-Simulator:
Charlotte lässt im Prozessor-Simulator selbst geschriebene Programme aus der Bsys1 Übung laufen, um ihren Lösungsansatz zu validieren. Um sich nicht zu langweilen, möchte sie gewisse Animationen, die sie bereits verstanden hat, überspringen oder schneller ausführen.

4.2 Use Cases

Anhand der Personas haben wir unsere Use Cases für den Prozessor-Simulator aus der Studienarbeit überarbeitet [1]. Im Anhang A befindet sich eine genauere Auflistung.

4.2.1 UC01 Einzelne Schritte des Prozessor-Zyklus sehen

“Als Benutzer möchte ich die einzelnen Schritte des Prozessor-Zyklus sehen, um besser zu verstehen, was im Prozessor genau passiert.”

Um diesen Use Case umzusetzen, ist es wichtig, dass man die Zustandsübergänge gut erkennt, um die Schritte auch richtig zu verstehen. Ausserdem muss man im Schritt “Execute Instruction” erkennen, dass die CPU die erhaltenen Daten verarbeitet.

4.2.2 UC02 ASM Instruktionen ausführen

“Als Benutzer möchte ich mein eigenes Programm eingeben und ausführen, um die verschiedenen Instruktionen besser zu verstehen.”

Für diesen Use Case muss eine Codeeingabe ermöglicht werden. Für den Benutzertyp Andreas sollen auch Beispielprogramme zur Verfügung stehen. Die Instruktionen, die in der Vorlesung Bsys1 verwendet werden, sind im Anhang B aufgelistet.

4.2.3 UC03 Ausführung steuern

“Als Benutzer möchte ich die Ausführung steuern können, um in meinem Tempo die Animationen zu sehen.”

Dieser Use Case ist für den Benutzertyp Charlotte relevant. Sie möchte vielleicht umfangreichere Programme ausführen und hat die Grundlagen schnell verstanden. Deshalb soll man das Animationstempo einstellen und wiederholende Schritte, wie zum Beispiel das Holen der Instruktion aus dem Memory, überspringen können.

4.3 Nicht-Funktionale Anforderungen

Die Nicht-Funktionalen Anforderungen an den Prozessor-Simulator aus der Studienarbeit haben wir ebenfalls überarbeitet [1]. Die genaue Auflistung ist im Anhang A ersichtlich.

Die Informatik Studierenden sollen die Applikation innerhalb von fünf Minuten verstehen und anwenden können. Ausserdem sollen vier von fünf Informatik Studierende nach einer 30-minütigen Verwendung der Applikation den Ablauf des Prozessor-Zyklus verstehen. Das Produkt kann ohne Installation in der neusten Version von Chrome auf Ubuntu Linux, MacOS, Windows genutzt werden.

Um einen schnellen und einfachen Austausch zu ermöglichen, sollen die Applikationskomponenten modular aufgebaut werden. Für die Performance sollen die Animationen flüssig sein und nicht stocken. Ausserdem soll sofort ein visuelles Feedback erscheinen, wenn ein Button gedrückt wird.

5 Lösungskonzept

Zu Beginn der Arbeit setzten wir uns damit auseinander, wie die noch nicht erfüllten und neu dazugekommenen Anforderungen an das Produkt umgesetzt werden können. Da die Umsetzung aller Ziele, den Rahmen der Bachelorarbeit sprengen würden, mussten wir ein paar davon weglassen oder vereinfachen. Wir trafen unsere Entscheidungen anhand der Richtlinie, dass das Endprodukt einsatzfähig und bedienbar sein soll.

5.1 Was fehlt zur Produktreife im Prototyp

Die Hauptanforderung der Bachelorarbeit ist, dass der Prototyp aus der Studienarbeit Produktreife erreicht. Wir haben uns deshalb Gedanken gemacht, was notwendig ist, um diese zu erreichen. Das wichtigste fehlende Feature, ist der Code Editor. Dort sollen die Studierenden ihren eigenen Code eingeben können, den sie anschliessend im Simulator ausführen können. Ausserdem fehlen die Status-Flags, die wir im Konzept angedacht haben, um noch näher an den Vorlesungsstoff zu kommen. Bei den Animationen wäre es schön, wenn klar ersichtlich wäre, was sich ändert, in dem bei jeder Änderung die entsprechenden Bytes kurz pulsieren. Dies haben wir so im Design-Ansatz angedacht. Ausserdem waren in der Studienarbeit die Buttons, die nur Icons angezeigt haben, unklar für die Usability Testpersonen. Damit unser Simulator gut bedienbar ist, sollen die Buttons deshalb beschriftet sein oder Tooltips erhalten. Das sind kleine Hinweistexte, die erscheinen, sobald man mit der Maus über ein Element fährt.

5.2 Weggelassene Funktionserweiterungen

Einige Funktionsanforderungen aus der Aufgabenstellung der Studien- und Bachelorarbeit und aus den Kickoff Meeting der Bachelorarbeit mussten wir weglassen, mit dem Ziel, dass der Simulator die Produktreife rechtzeitig erreicht. In der Aufgabenstellung wurden die Stabilität und Benutzbarkeit als wichtiger bewertet als der Funktionsumfang. Die in den folgenden Abschnitten beschriebenen Funktionalitäten wären alle aufwändig gewesen, da sie nicht im Konzept der Studienarbeit vorhanden waren, und sie somit vor der Umsetzung auf die Verständlichkeit hätten geprüft werden müssen. Ausserdem hätten diese unserer Meinung nach zu wenig Mehrwert für die Studierenden geboten. Wir haben während der Arbeit jedoch darauf geachtet, dass es in einer Folgearbeit möglich wäre, die weggelassenen Funktionalitäten zu ergänzen. Ein Beispiel für eine Funktionalität aus der Aufgabenstellung, die nur wenig Mehrwert für die Studierenden bietet, sind die verschieden festlegbaren Speicherbereiche.

Die Instruktionen betreffen meistens nur Daten an einer Stelle im Memory, somit ist die Darstellung eines einzigen scrollbaren Speicherbereichs ausreichend. Deshalb haben wir uns von Anfang an entschieden die verschiedenen Speicherbereiche nicht umzusetzen.

Auf der Abstraktionsstufe unseres Konzepts, dass wir bereits in der Studienarbeit entwickelt haben, fehlt der Speicherbus. Er liegt konzeptionell zwischen der CPU und dem Memory. Die CPU greift über ihn auf den Speicher zu, indem sie ihm die Adresse übergibt und er ihr die gewünschten Daten aus dem Memory zurückliefert. Er gehört zum Inhalt, der in der Vorlesung vermittelt wird und sollte deshalb im Simulator angezeigt werden. Wir mussten ihn jedoch leider aus Zeitgründen weglassen. Anhand unseres Design Ansatzes der Studienarbeit, wäre es jedoch möglich zu erklären, wo der Speicherbus liegt und was seine Aufgabe ist.

Bereits in der Studienarbeit haben wir erkannt, dass wir den Schritt “Get Instruction” zu stark vereinfacht haben. Die CPU liest dort in Wirklichkeit nicht die ganze Instruktion auf einmal, sondern Byte für Byte. Um dies richtig umzusetzen, wäre es wünschenswert zu sehen, wie die CPU zuerst den Opcode liest und anhand von diesem erkennt, wie viele Bytes zur Instruktion gehören. Wir haben diese Funktionalität als zu aufwändig und für zu wenig wichtig für die Studierenden empfunden.

Wie in unserem Design-Ansatz aus der Studienarbeit angedacht, wollten wir zur Reduzierung der Informationsüberlastung die Bytes, die nicht verwendet werden, leicht ausblenden. Sobald aber auf sie zugegriffen wird, zum Beispiel wenn sie gelesen werden, sollen sie eingeblendet werden. Im Kickoff Meeting wurde zusätzlich gewünscht, dass wir ein Fading anbieten. Die Bytes würden somit nach dem Zugriff Schritt für Schritt ausgeblendet, bis sie wieder unbenutzt aussehen. Wir finden diese Idee gut, sie bringt jedoch nur einen Mehrwert, wenn die Studierenden riesige Programme laufen lassen und zu viele Bytes eingeblendet werden. Deshalb haben wir diese Idee mit einer tieferen Priorität eingestuft und nicht umgesetzt.

5.3 Usability Tests

In unserem Projekt sind die Benutzbarkeit und Verständlichkeit sehr wichtig. Deshalb haben wir uns zu Beginn entschlossen zwei Usability Tests im Verlaufe des Projekts durchzufüh-

ren. Das erste planten wir in der Mitte des Projekts, um allfällige Fehler im Simulator frühzeitig zu erkennen und diese noch bis zum Abschluss der Arbeit zu beheben. Das zweite haben wir gegen Ende der Arbeit geplant. Das Ziel dieses Tests war, herauszufinden, ob unser Endprodukt die Erwartungen der Studierenden erfüllt und im Unterricht eingesetzt werden kann.

5.3.1 Erster Usability Test

Im ersten Test in der Mitte des Projekts, wollten wir sehen, wie die Benutzenden den Simulator verwenden und ob sie Schwierigkeiten mit der Bedienung haben. Ausserdem sollte so die Verständlichkeit der Animationen und des Vorlesungsinhalts sichergestellt werden. Durch die Resultate des Tests erhofften wir uns die Features so priorisieren zu können, dass das Endprodukt einen möglichst grossen Mehrwert für die Studierenden bringt. Um die Verständlichkeit des Prozessor-Zyklus im Simulator zu testen, haben wir uns entschieden Testpersonen zu suchen, die die Bsys1 Vorlesung nicht mehr so präsent oder nur wenig Informatikkenntnisse haben.

5.3.2 Zweiter Usability Test

Wir haben für den zweiten Usability Test zum Abschluss dieser Arbeit geplant, als Testpersonen Studierende zu nehmen, die die Bsys1 Vorlesung möglichst präsent haben, um zu sehen, ob unser Simulator ihrer Meinung nach zur Vorlesung passt. Ausserdem wollten wir sehen, ob wir unsere Ziele zur Benutzbarkeit und Verständlichkeit aus den nicht-funktionalen Anforderungen erreicht haben. Da die Vorlesung Bsys1 im Frühlingssemester nicht angeboten wird, haben wir uns entschieden Studierende aus dem 2. Semester anzufragen, die das Modul “Betriebssysteme 2” (Bsys2) besuchen. Um einen Vergleich von unserem Simulator zu bestehenden Simulatoren zu haben, wollten wir die Studierenden ein Programm in einem fremden Simulator ausführen lassen, ohne zu erwähnen, dass es sich dabei nicht um unseren handelt. Sie sollen uns dann erklären, was passiert ist und Feedback zum Design und zur Benutzbarkeit geben. Erst danach sollen sie unseren Simulator verwenden und dasselbe Programm noch einmal ausführen.

5.4 GPL Lizenz

Da das Endprodukt als Open Source Software veröffentlicht werden soll, mussten wir uns zu Beginn der Arbeit entscheiden, unter welcher Lizenz wir sie veröffentlichen wollen. Dabei war es wichtig, dass diese mit den Lizenzen, die im Prototyp verwendet werden, kompatibel

ist. Die Unicorn Engine unterliegt der GNU General Public License Version 2 (GPLv2) ohne die Möglichkeit ein Upgrade zur Version 3 zu machen. Bei GPLv2 handelt es sich um eine Lizenz für freie Software, die sicherstellt, dass man Kopien der Software verbreiten kann, aber auch das Recht hat, den Quellcode dieser Software zu erhalten.

Diese Lizenz ist jedoch sehr restriktiv. Man darf zwar die Software ändern oder Teile davon übernehmen, muss dann jedoch seinen selbst geschriebenen Code ebenfalls unter einer mit GPLv2 kompatiblen Lizenz veröffentlichen. Da wir die Unicorn Engine verwenden, unterliegt unser Prozessor-Simulator ebenfalls dieser Restriktion. Der Nachteil, wenn unser Code der GPLv2 unterliegt, ist, dass man in einer Folgearbeit zwar die Unicorn Engine ersetzen kann, den eigenen Code aber trotzdem unter einer zu GPLv2 kompatiblen Lizenz veröffentlichen muss. [2]

Eine Apache Lizenz wäre in dieser Situation besser geeignet. Diese ermöglicht, dass man Änderungen und den eigenen Code unter einer anderen Lizenz veröffentlichen darf, sofern die Rechte und Pflichten erfüllt sind [3]. Die Apache Lizenz ist zwar mit GPLv3, aber nicht mit GPLv2 kombinierbar. GPLv2 würde je nach dem ein Upgrade auf GPLv3 erlauben [4], Unicorn verwendet jedoch QEMU als Emulator, der die Verwendung späterer Versionen als GPLv2 explizit untersagt.

Um die Unicorn Engine verwenden zu dürfen, bleibt uns nichts anderes übrig, als unseren Code ebenfalls unter GPLv2 zu veröffentlichen. Falls der Emulator in einer Folgearbeit durch einen selbst programmierten oder anders lizenzierten Code ersetzt werden würde, ziehen wir in Betracht, unseren Code unter einer anderen Lizenz noch einmal zu veröffentlichen.

5.5 Nasm Syntax

Nachdem die wichtigsten Entscheide gefällt waren, entwickelten wir ein Konzept, wie die fehlenden Funktionalitäten im Simulator umgesetzt werden können. Wir starteten dazu bei den Libraries, da diese einen grossen Einfluss auf die Architektur haben.

Im Konzept unserer Studienarbeit war geplant, den Keystone Assembler[5] für unseren Simulator zu verwenden. Als wir jedoch damit erste Code Beispiele aus der Vorlesung assemb-

[5] Keystone: https://www.keystone-engine.org/

lierten, bemerkten wir, dass dieser das Setzen von lokalen Variablen und Labels nicht unterstützt. Das ist ein grosses Problem, da diese Instruktionen in den meisten Programmen aus den Übungen verwendet werden. Den Studierenden würde so verunmöglicht, den in der Vorlesung gelernten Code einzugeben, was unseren ganzen Simulator für den Unterricht ungeeignet machen würde. Deshalb entschieden wir uns den Netwide Assembler[6] (Nasm) einzubinden, der auch in der Vorlesung genutzt wird.

Ausserdem fiel uns auf, dass die Syntax des Capstone-Disassemblers nicht mit der in der Vorlesung verwendeten Nasm Syntax übereinstimmt. Eine Instruktion, die sich im Prototyp des Simulators stark zur Vorlesung unterscheidet, ist die Move-Instruktion. Zum Beispiel wird "mov rax, [0x0]" in Capstone als "mov rax, qword ptr [0]" dargestellt. Unserer Meinung nach erschwert dies den Studierenden, den Zusammenhang zwischen den Instruktionen in der Vorlesung und unserem Simulator zu erkennen. Deshalb war es uns wichtig, diese Syntaxen aneinander anzugleichen. Um den Studierenden das bestmögliche Produkt zu bieten, suchten wir nach einem Disassembler, der die Nasm Syntax anbietet und stiessen auf den Netwide Disassembler[7] (Ndisasm), der Disassembler von Nasm selbst.

Das grösste Problem bei der Einbindung der beiden Libraries war, dass es sich zwar um Multiplattform-C-Libraries handelte, aber keine direkte Portierung nach JavaScript zur Verfügung stand. In der Studienarbeit setzten wir Unicorn.js[8] ein, das die C-Library Unicorn nach JavaScript portiert. Durch die Modifikation des Build File erhielten wir damals Einblick, wie man so eine Portierung machen könnte und entschieden uns nun, Nasm auf dieselbe Weise einzubinden. Diese Entscheidung war sehr riskant, da die Portierung von Libraries eine schwierige Aufgabe ist, die unvorhergesehene Probleme mit sich bringen kann. Aber in unserem Fall überwog der Vorteil, das gleiche Tool wie im Unterricht zu verwenden, dem Nachteil, Zeit zu verlieren, die man für etwas anderes hätte verwenden könnte.

5.6 Architektur

Die Architektur des Prototyps (siehe Abbildung 4) wollten wir mit den neuen Libraries und dem Code Editor folgendermassen aufbauen. Sie ist in zwei Hauptkomponenten, den Editor und Simulator, aufgeteilt. Im Editor gibt es eine Vue-Komponente, wo der Benutzende seinen Assembly Code eingeben kann. Dieser wird beim Start des Programms vom Nasm in

[6] NASM: https://www.nasm.us/
[7] NDISASM: https://www.nasm.us/doc/nasmdoca.html
[8] Unicorn.js: https://alexaltea.github.io/unicorn.js/

Maschinencode umgewandelt und mit Hilfe der URL an die Vue-Komponente des Simulators übergeben. Zusätzlich sind im Editor Service einige Beispielprogramme hinterlegt, die im Editor angezeigt werden.

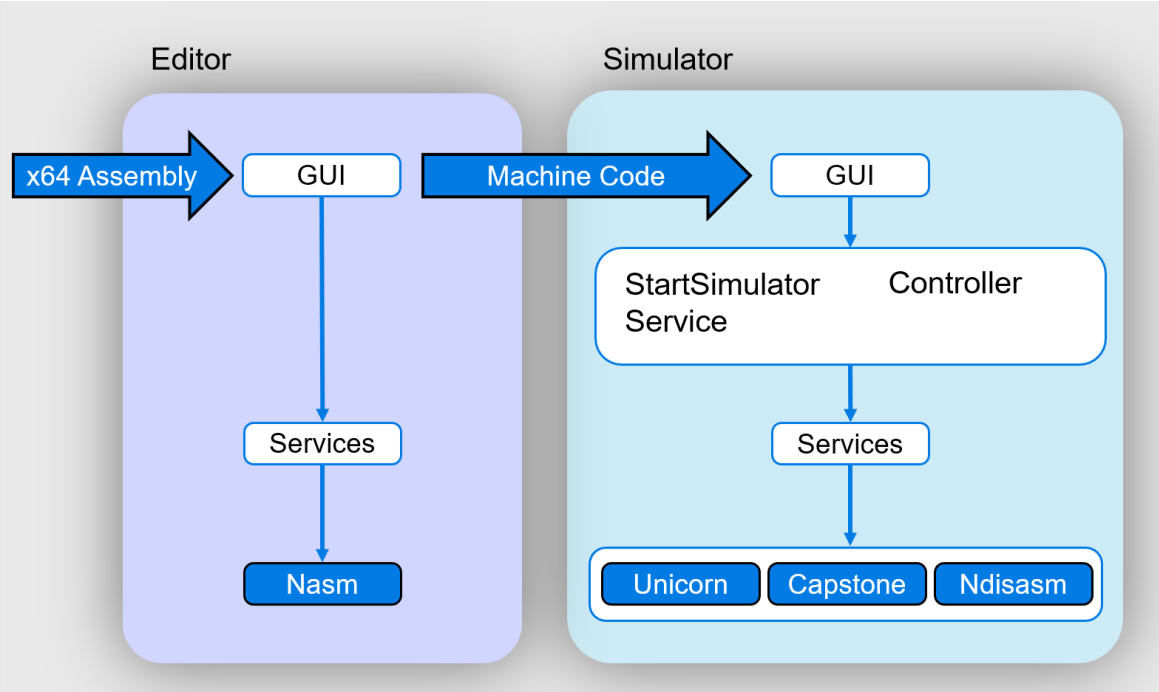


Abbildung 4 Architekturübersicht

Die Vue-Komponente **Simulator** erhält den Maschinencode vom **Editor**. Um das Memory und die CPU mit den richtigen Daten zu laden, ruft die Komponente als erstes den **StartSimulatorService** auf. Dieser startet eine Unicorn Emulator Instanz und lässt diese den Maschinencode ins Memory laden. Ausserdem startet der Service eine Capstone Disassembler-Instanz. Der Service liefert ein **Program** Objekt an das GUI aus, das die Konfiguration und die Unicorn und Capstone Instanz enthält (siehe Abbildung 5).

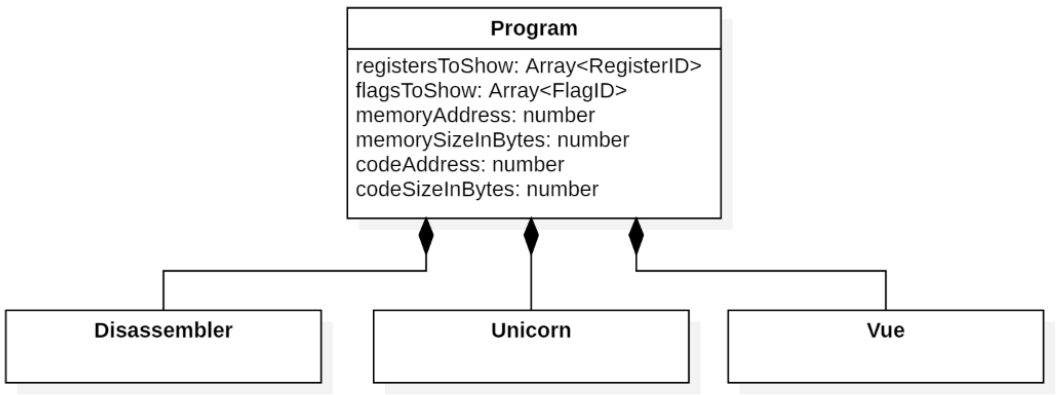


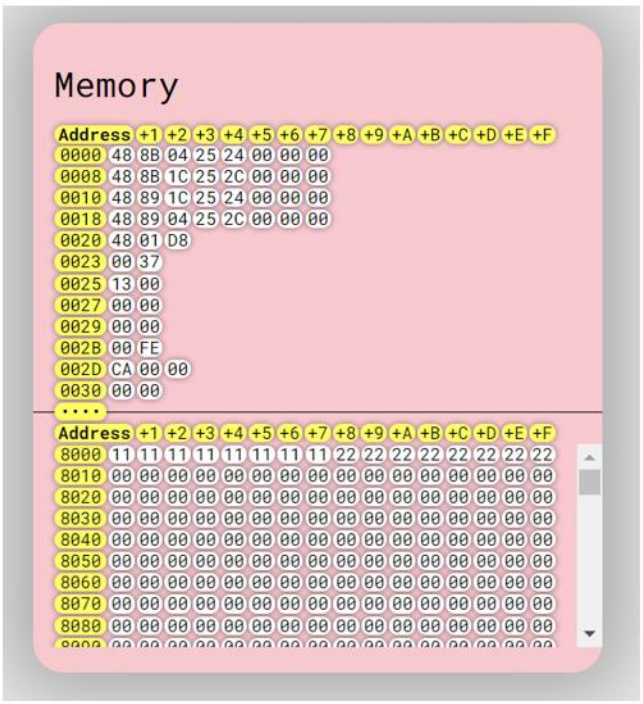
Abbildung 5 Program Objekt

Mit dem erhaltenen **Program** Objekt startet die **Simulator** Vue-Komponente den **Controller**. Um die Daten aus dem Emulator zu erhalten, ruft der **Controller** die Services auf, die ein **State** Objekt erstellen, das die Daten des aktuellen Zustands aus dem Emulator enthält. Mit dem **State** Objekt kann der **Simulator** dann seine untergeordneten Vue-Komponenten, beispielsweise das Memory oder die CPU mit Daten versorgen. Nun wird der Simulator im GUI angezeigt.

Sobald der Benutzende im GUI auf “Next Step” klickt, ruft die Simulator-Komponente den **Controller** erneut auf. Der weiss, in welchem Schritt man sich aktuell befindet und startet dementsprechend in den **AnimationServices** die passenden Animationen. Im Schritt “Get Instruction” lesen die **DataServices** die nächste Instruktion aus dem Unicorn-Emulator aus und lassen sie von ihm ausführen. Ausserdem holen sie sich die Daten zur Instruktion aus den beiden Disassemblern Capstone und Ndisasm. Bei “Execute Instruction” und “Increment Instruction Pointer” werden dann die vorher erhaltenen Daten aktualisiert.

5.7 Darstellung Instruktionen im Memory

In der Studienarbeit überlegten wir uns, wie wir die Instruktionen darstellen können, so dass klar ist, dass sie im Memory liegen. Wir haben uns entschieden, die Instruktionen separat in einem eigenen Bereich anzuzeigen und jede Instruktion auf einer eigenen Zeile darzustellen. Durch die Verwendung des Nasm wird es jedoch möglich mit Funktionen wie zum Beispiel “dq” Variablen zu setzen. Die Werte der Variablen werden dem Maschinencode direkt angehängt. Ein Beispiel ist in der Abbildung 6 ersichtlich. Wenn wir diesen ganzen Maschinencode an Capstone übergeben, versucht er diesen zu interpretieren und liefert die erhaltenen Instruktionen zurück. Er stoppt sobald keine Bytes mehr zur Verfügung stehen oder diese keiner validen Instruktionen entsprechen. Das heisst je nachdem ob man Werte als Variablen verwendet, die als Instruktion interpretiert werden können, bricht Capstone früher oder später ab. Somit ist es immer unterschiedlich, ob er die Variablen als Instruktionen anzeigt oder nicht, deshalb finden wir unsere Darstellung nicht mehr optimal.




```
1   BITS 64
2
3   SECTION .data
4   firstVar:
5       dq 0 x1337
6   otherVar:
7       dq 0 xcafe
8
9   SECTION .text
10  global _start
11  _start:
12      mov rax, [ firstVar ]
13      mov rbx, [ otherVar ]
14      mov [ firstVar ], rbx
15      mov [ otherVar ], rax
16      add rax, rbx
```

Abbildung 6 Darstellung eines Programms mit Variablen (rechts) im Prototypen

Ein weiterer Nachteil unserer Darstellung ist, dass wir den Maschinencode als Ganzes im Voraus an Capstone übergeben müssen. Somit wird nicht unterstützt, dass man bei einem Call oder Jump zu einer Adresse ausserhalb des Codebereichs springen kann. Der Code an dieser Adresse gehört nicht zum Teil des zu Beginn eingegebenen Maschinencodes. Es wäre also geeigneter, wenn wir Capstone im laufenden Programm immer nur eine Instruktion aus dem Memory lesen lassen und nicht den gesamten Maschinencode. Capstone könnte uns dann die Information geben, welche Bytes zur aktuellen Instruktion gehören. Diese können wir dann Unicorn übergeben und dort ausführen. Somit könnte man auch im Programm dynamisch Änderungen am Maschinencode vornehmen und Capstone würde die aktualisierte Instruktion lesen.

Um dies umzusetzen, überlegten wir uns den Instruktionen-Block im Memory in den unteren Speicherbereich zu integrieren. Somit wird der Maschinencode genau wie die restlichen Daten im Speicher dargestellt. Der Vorteil bei dieser Darstellung ist, dass sie näher an der Realität liegt und die Studierenden noch besser erkennen können, dass die CPU den Code aus dem Memory liest. Der Nachteil ist, dass der Maschinencode und die Speicherbereiche, die einem interessieren, weit auseinander liegen können, so dass nicht mehr beides gleichzeitig ersichtlich ist und man hin und her scrollen muss.

Ein weiterer Nachteil, wenn die Instruktionen nicht mehr einzeln dargestellt werden, ist, dass die Darstellung der Pfeile, die wir in der Studienarbeit angedacht haben, nicht mehr funktioniert. Der Instruction-Pointer sollte als Pfeil vom CPU-Feld “Instruction Pointer” zur Zeile der entsprechenden Instruktion zeigen (siehe Abbildung 7). Liegen die Instruktionen nebeneinander im Memory, wird es schwierig, den Pfeil auf die aktuelle Instruktion zu richten.

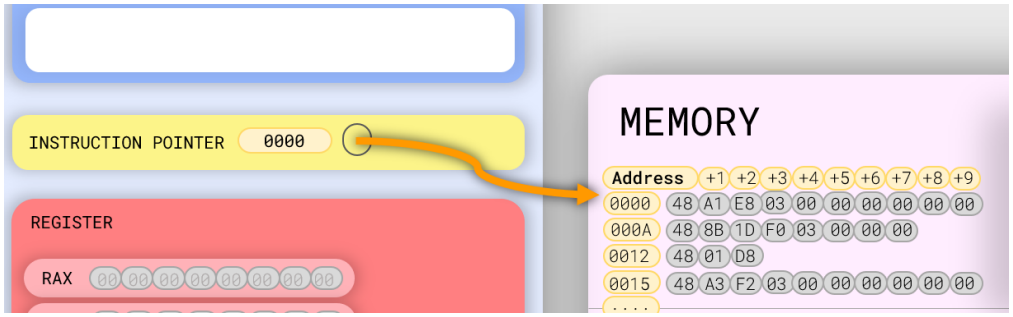


Abbildung 7 Darstellung Instruction Pointer im Designansatz der Studienarbeit

Ausserdem wollten wir zusätzlich zum Instruction-Pointer weitere Pfeile anzeigen für die visuelle Unterstützung der Byte Bewegungsanimationen. Diese wollten wir zwischen den Operanden und den Werten anzeigen, wenn die Instruktion ausgeführt wird. So sollte man erkennen, von wo die Daten gelesen und wohin sie geschrieben werden. Wenn man die Instruktionen nicht mehr separat darstellt, zeigen mehrere Pfeile ins Memory und überkreuzen sich je nachdem, was zu Verwirrung zwischen den verschiedenen Pfeilen führen könnte.

Durch das scrollbare Memory wäre eine Darstellung von Pfeilen sowieso schwierig geworden. Die Pfeile hätten mitscrollen müssen, solange das Element, auf das sie zeigen, im Memory ersichtlich ist. Sobald das Element nicht mehr sichtbar ist, hätte man dann die Pfeile je nachdem oben oder unten am Memory Rand darstellen müssen. Da man sich konstant über die Scrollposition hätte informieren müssen, befürchteten wir, dass diese Implementation der Performance schaden könnte. Da unsere Animationen auch ohne Pfeile funktionieren, haben wir uns dann entschieden, jegliche Pfeile wegzulassen.

Um im Memory trotzdem ersichtlich zu machen, wo der Instruction-Pointer hinzeigt, hatten wir die Idee, die aktuelle Instruktion im Memory entsprechend dem Feld “Instruction Pointer” in der CPU gelb einzufärben (siehe Abbildung 8). Das Byte, auf das der Instruction-Pointer zeigt, soll sich zusätzlich von den anderen abheben. Da die noch nicht verwendeten Bytes anders dargestellt werden, sieht man ausserdem, wo sich der Maschinencode-Bereich befindet.

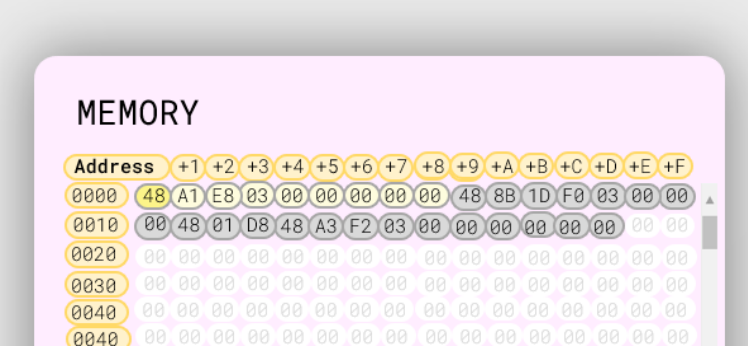


Abbildung 8 Neuer Designansatz zur Darstellung des Instruction Pointer im Memory

Damit die Animationen trotz der verteilten Bereiche im Memory funktionieren und verständlich sind, haben wir uns entschieden jeweils vor dem animierten Zugriff auf ein Byte, automatisch an die richtige Position im Memory zu scrollen. Die Positionsveränderung soll so animiert werden, dass sie von den Benutzenden visuell mitverfolgt werden kann. Das verhindert, dass das Memory von einer Position zur anderen springt und der Benutzende nicht sieht was passiert ist und somit die Orientierung verliert.

5.8 Animation der Instruktionen

Zu Beginn dieser Arbeit hatten wir die Absicht, die “mov” Instruktion, die wir in der Studienarbeit geplant haben, fertig umzusetzen. Um zu zeigen, dass die CPU nicht nur Daten verschiebt, sondern diese auch verarbeitet, wollten wir zudem noch eine “add” Instruktion anbieten und die Berechnung der Addition zeigen. Bei allen anderen Instruktionen wären dann die Daten ohne eine Animation aktualisiert worden. Wir haben uns deshalb überlegt, wie wir ein “add” Instruktion animieren können. Dabei ist uns aufgefallen, dass es zwischen den meisten Instruktionen viele Parallelen gibt. Die CPU liest oftmals Daten ein, verarbeitet diese und schreibt sie an einen Zielort. Da kam uns die Idee, dass wir statt jede Instruktion einzeln auch alle generisch animieren könnten. Somit würden für jede Instruktion jeweils die Lese- und Schreibzugriffe der CPU gezeigt. Dieses Verfahren bringt einen grossen Mehrwert für den Simulator, da so alle Instruktionen animiert werden können.

Ausserdem machten wir uns in der Studienarbeit Gedanken, wie wir die verschiedenen Instruktionen und deren Operanden erkennen könnten. Durch die generische Animation fällt dieses Problem weg und wir müssen nur wissen, welche Daten die CPU liest und welche sie schreibt. In einer Folgearbeit wäre trotzdem denkbar, dass man für gewisse Instruktionen,

beispielsweise “jmp”, dedizierte und detaillierte Animationen anbietet. Ausserdem könnte man zwischen dem Lesen und Schreiben der Werte beispielsweise ein Popup anbieten, dass zeigen würde, wie die CPU-Werte bei einem “add” addiert.

Neben den generisch animierten Lese- und Schreibzugriffen sollte man sehen können, dass die Daten von der CPU auch verarbeitet werden. Wir haben zusammen mit Prof. Stefan Richter und Florian Bruhin Ideen gesammelt. Zum Beispiel könnte man eine Kurbel darstellen, die gedreht wird, oder eine kleine Person oder einen Roboter, der die Daten verarbeitet. Da man nicht direkt sieht was passiert, wäre auch ein Zauberstab eine Option.

Da die Darstellung im Rahmen der Bachelorarbeit umsetzbar sein soll, muss sie jedoch möglichst einfach sein. Wir haben uns entschieden iterativ vorzugehen und in der ersten Iteration die Input-Daten in der CPU wie in einem schwarzen Loch verschwinden zu lassen. Die Output-Daten würden dann wieder aus der CPU hervortreten und an ihr Ziel fliegen. Diesen Ansatz wollten wir im ersten Usability Test überprüfen und so Probleme frühzeitig zu erkennen und zu entscheiden, wie wir fortfahren.

5.9 Darstellung Stack

Ein grosser Wunsch von den Bsys1 Studierenden, die am Usability Test in der Studienarbeit teilgenommen haben, war die Darstellung des Stacks. Wir haben uns überlegt, wie wir diesen abstrahieren könnten, um ihn in unserem Simulator einzubauen. Unsere erste Idee war, den Stack als Stapel in einer eigenen Komponente darzustellen. Man hätte bei einem “push” eine Schicht auf den Stapel gelegt und bei “pop” die oberste Schicht wieder weggenommen.

Diese Idee haben wir verworfen, weil wir den Studierenden zeigen möchten, dass der Stack im Memory liegt. Um das zu erreichen, wollen wir analog zum Instruction-Pointer, den Stack-Pointer darstellen, der auf das im Stack zuletzt abgelegte Element im Memory zeigt. Ausserdem soll ersichtlich sein, dass der Stack rückwärts wächst. Bei “push x” wird also der Stack-Pointer um acht verringert und “x” an die neue Position kopiert. Bei “pop x” werden acht Bytes von der Adresse, wo der Stack-Pointer hinzeigt, an die Stelle “x” kopiert und der Stack-Pointer um acht erhöht. Des Weiteren gibt es den Base-Pointer, den wir auf die gleiche Weise darstellen wollen. Der Base-Pointer ist beim Aufrufen von Funktionen von Bedeutung. Um in der Funktion auf Argumente zuzugreifen, die man vor dem Funktionsaufruf auf

den Stack gepusht hat, setzt man den Base-Pointer auf die aktuelle Position des Stack-Pointers. So kann man über ihn auf die Argumente zugreifen und gleichzeitig den Stack weiterverwenden. [5]

Wir haben in Microsoft PowerPoint einen ersten Design Ansatz entwickelt, den wir im Verlauf der Arbeit umsetzen wollten (siehe Abbildung 9). Die Stack- und Base-Pointer werden wie der Instruction-Pointer in eigenen Komponenten dargestellt und erhalten jeweils eine eigene Farbe. Die zuletzt auf den Stack gelegten Bytes werden in der Farbe der Pointer eingefärbt. Das Byte, auf das der Pointer zeigt, ist zusätzlich noch dick umrahmt.

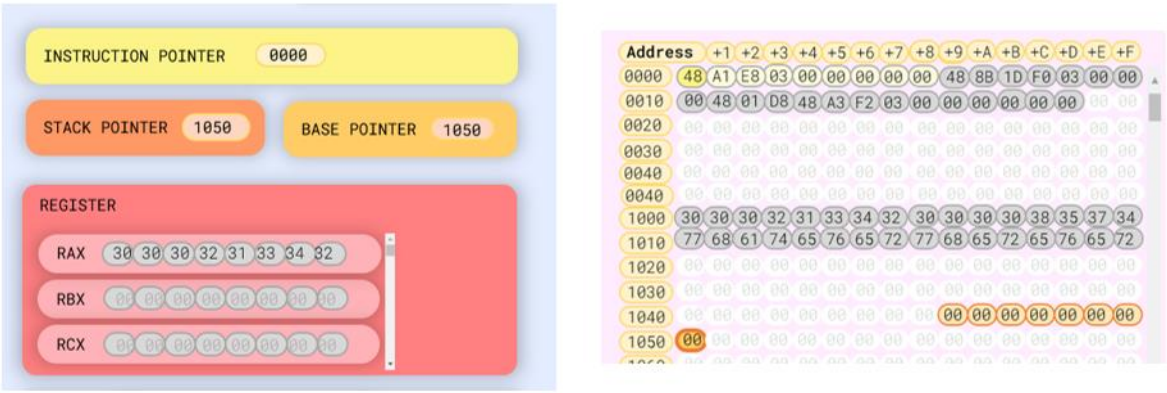


Abbildung 9 Designansatz zur Darstellung des Stack- und Base-Pointers

5.10 Darstellung des Bytes in verschiedenen Zuständen

Für die neue Idee, die Bytes im Memory zu markieren, wenn sie zum Stack oder zur Instruktion gehören, benötigen wir einen Mechanismus, um diese einzufärben. Zudem haben wir in unserer Studienarbeit herausgefunden, dass man die Informationsüberlastung reduzieren kann, indem man die Bytes im Memory, die nicht verwendet werden, leicht ausblendet. Die unbenutzten Bytes sind ausserdem bei der Ausführung des Programms von Vorteil, um das Ende eines Programms zu finden. Bisher wurde das Ende des Programms dann erreicht, wenn der Instruction-Pointer auf ein Byte zeigte, das nicht zum original eingegebenen Code gehörte. Wenn man also während der Laufzeit eine Instruktion an den Code angehängt hat, wurde diese nicht mehr ausgeführt. Durch die Markierung der verwendeten Bytes könnte man definieren, dass die Ausführung ausserhalb des Codebereichs fortgesetzt wird, wenn der Instruction-Pointer auf ein solches Byte zeigt.

Um die Bytes unterschiedlich darzustellen, benötigen wir einen Mechanismus, um der **Byte-Vue** Komponente von den Services aus mitzuteilen, in welchem Zustand sich das Byte befindet. Ein Byte kann entweder benutzt oder unbenutzt sein. Bei den benutzen Bytes sollte es zudem später möglich sein ein Fading hinzuzufügen, das die Bytes Schritt für Schritt wieder ausblendet. Ein Byte kann ausserdem zur aktuellen Instruktion oder einem Element auf dem Stack gehören, wobei beim Stack nur die an den Base- und Stack-Pointer anliegenden Bytes eingefärbt werden. Wir haben zwei verschiedene Ansätze, wie man das umsetzen könnte.

5.10.1 Möglichkeit 1: Dynamisch CSS-Klassen hinzufügen

Eine Möglichkeit wäre den Bytes, wie bei den Animationen, dynamisch CSS-Klassen hinzuzufügen und wieder wegzunehmen. Wenn ein Byte beispielsweise Teil der acht oberen Bytes im Stack wäre, würde man mit seiner **LocationId** auf das HTML-Element zugreifen und ihm eine Klasse “stackByte” hinzufügen. Man müsste jedoch bei jedem Stack Update herausfinden, welche Bytes im Moment eine Klasse “stackByte” haben und diese bei allen entfernen, bevor man den nächsten acht Bytes die Klasse setzt. Dies führt zu einem grösseren Programmieraufwand in den Services.

Für das Fading müsste man wahrscheinlich pro Ausblendungsstufe eine CSS-Klasse erstellen. In den Services müsste man in jedem Schritt die Bytes der jeweiligen Stufen suchen und eine Stufe höher setzen, damit sie etwas mehr ausgeblendet werden. Dies könnte sehr aufwendig und fehleranfällig sein.

Um herauszufinden ob der Instruction-Pointer auf ein unbenutztes Byte zeigt und das Programm beendet ist, müsste der **Controller** auf das GUI zugreifen, um die CSS-Klasse des betroffenen Bytes auszulesen. In der jetzigen Architektur greift der **Controller** jedoch nie direkt auf das GUI zu, sondern ändert die Daten im **State** Objekt. Es wäre unschön, wenn man nun einen anderen Weg einschlagen würde.

5.10.2 Möglichkeit 2: Informationen im State Objekt mitliefern

Eine andere Möglichkeit wäre im **State** die Information mitzugeben, welche Bytes welche Eigenschaften haben. Dazu würde man dem **State** Objekt ein Attribut **ByteInformation** hinzufügen, in dem man die Information hinterlegt, welche Bytes im Moment beispielsweise zum Stack gehören und eingefärbt werden. Um die Daten zu aktualisieren, könnte man in den

Services die entsprechenden Attribute im **ByteInformation** Objekt mit den neuen Werten ergänzen oder komplett ersetzen.

Man müsste jedoch in der **ByteVue** Komponente die **LocationId** des aktuellen Bytes mit denjenigen in der **ByteInformation** abgleichen, um herauszufinden, welche Darstellung verwendet werden soll. Der grössere Programmieraufwand der ersten Möglichkeit würde hier von den Services in die Vue-Komponenten verschoben. Ausserdem könnte bei dieser Möglichkeit die Performance leiden. Wir haben im Memory 4095 Bytes und bei jeder Änderung müsste jedes Byte seine **LocationId** mit dem **ByteInformation** Objekt abgleichen.

Bei den benutzten und unbenutzten Bytes haben wir uns deshalb überlegt, ein Attribut "isUsed" direkt im **Byte** Objekt abzuspeichern, anstatt die Information im **ByteInformation** Objekt mitzugeben. Dies hätte das Performance Problem etwas reduzieren können. Das Problem wäre jedoch, dass wir, um das Memory zu aktualisieren, jeweils die Funktion "**getMemory**" aufrufen, die das komplette Memory ausliest und in Bytes abfüllt. Dadurch werden die Bytes jedes Mal neu erstellt und die Information "isUsed" würde zurückgesetzt werden. Es wäre ein grosser Programmieraufwand angefallen, um die Memory Aktualisierung neu zu implementieren. Wir haben diese Idee deshalb verworfen.

5.10.3 Entscheidung

Das Fading wäre mit der zweiten Methode einfacher umzusetzen als mit der ersten. Man würde im Objekt beispielsweise verschiedene Arrays pro Ausblendungsstufe führen. So könnte man in jedem Schritt die Elemente eines Arrays, in den nächsten Array weitergeben, um die Bytes eine Stufe stärker auszublenden. Auch das Bestimmen des Endes des Programms wäre einfacher und schöner in der zweiten Methode, da der **Controller** nur auf das **State** Objekt zugreifen müsste und nicht auf das GUI. Wir haben uns deshalb für die zweite Variante mit dem **ByteInformation** Objekt entschieden und nahmen dafür ein Performance Problem in Kauf.

5.11 Darstellung der Flags

Im Konzept der Studienarbeit war die Darstellung der Status-Flags angedacht. Die Flags, die in Bsys1 verwendet werden, sind das Carry Flag (CF), Zero Flag (ZF), Sign Flag (SF) und das Overflow Flag (OF). Sie zeigen je nach Status eins oder null an. Wenn zum Beispiel das

Resultat einer Instruktion null ist, wird das Zero Flag gesetzt. Die Flags befinden sich bitweise codiert im Register EFLAGS. Deshalb wollten wir die Flags auch bei den Registern darstellen. Wir haben uns zusammen mit Prof. Stefan Richter und Florian Bruhin Gedanken gemacht, wie wir die Flags darstellen könnten. Die daraus entstandenen Ansätze sind in der Abbildung 10 dargestellt.

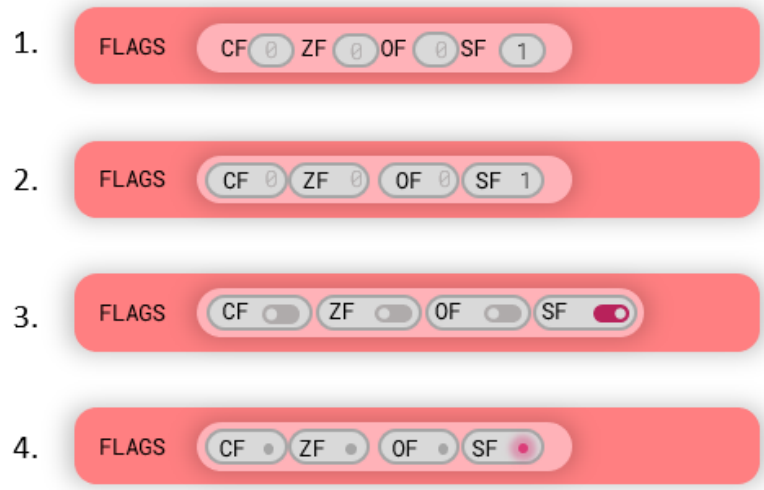


Abbildung 10 Verschiedene Designansätze für die Darstellung der Flags

Die erste und einfachste Variante wäre, dass die Werte der Flags in den bereits verwendeten **ByteVue** Komponenten dargestellt werden und daneben der Name stehen würde. Für Animationen wäre es jedoch besser, wenn die Flags, so wie in der zweiten Variante dargestellt, zusammen mit dem Namen bewegt würden, damit man erkennt, welcher Wert gelesen wird. Um zu zeigen, dass die Flags nur zwei Zustände haben, könnte man auch Toggle Buttons verwenden wie in Variante drei. Dort könnte jedoch das Problem auftreten, dass diese dem Benutzenden signalisiert, dass man dort den Wert ändern könnte. Unsere letzte Idee wäre mit einer Art LED zu signalisieren, ob das Byte gesetzt wurde. Aufgrund der Accessibility müsste man aufpassen, dass die beiden Zustände auch für Personen mit einer Sehschwäche unterscheidbar sind.

Da für uns keine der Varianten überzeugend war, haben wir uns entschieden im ersten Schritt die erste und einfachste Variante umzusetzen. So konnten wir uns auf die Zustandsänderungen und Animationen der Flags fokussieren. Diese sind am wichtigsten für das Verständnis. Falls in einer Folgearbeit ein besseres Design entwickelt werden würde, könnte unsere simple Darstellung problemlos überarbeitet werden.

5.12 Alle Register anzeigen

Eine Anforderung ist, dass man alle Register aus der Vorlesung im Simulator sehen kann. Da es sich dabei um ungefähr 18 Register handelt, finden wir die Informationsüberlastung zu gross. Wir haben uns entschieden im Simulator nur jeweils die Register anzuzeigen, die im Programm verwendet werden. Sobald eine Instruktion gelesen wird, werden somit Register, die noch nicht angezeigt werden zur Ansicht im Simulator hinzugefügt. Falls viele Register verwendet werden, möchten wir den Registerbereich analog zum Memory scrollbar machen. Im Schritt “Execute Instruction” überlegten wir zusätzlich alle Register auszublenden, die während der Instruktionsausführung nicht verwendet werden. Das könnte jedoch verwirrend sein, deshalb entschieden wir uns, die Register nicht mehr auszublenden, sobald sie einmal eingeblendet sind.

Die sogenannten Long-Size-Register sind acht Bytes gross und haben verschiedene Unterbereiche. Zum Beispiel entspricht das Int-Size-Register EAX der Hälfte der Bytes des Long-Size-Registers RAX (siehe Abbildung 11). Es wurde die Anforderung an uns gestellt, dass die Verschachtelung der verschiedenen Register im Simulator ersichtlich sein soll.

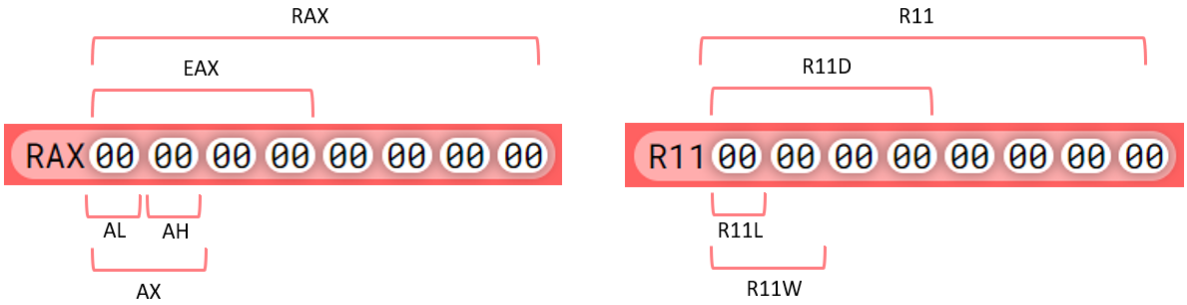


Abbildung 11 Register RAX und R11 und ihre Unterregister

Wir haben nach Ideen gesucht, wie dies gut dargestellt werden könnte. Zum Beispiel könnte man ein Raster über die Register legen und die verschiedenen Bereiche markieren. Da sich die Benennung der Unterregister zwischen den Registern stark unterscheidet, müsste man jedes Register einzeln beschriften, statt eine einzige Beschreibung für alle Register anzuzeigen.

Eine weitere Idee wäre, nur den Startpunkt des jeweiligen Unterregisters zu beschriften. Das funktioniert bei Registern wie RAX jedoch nicht, da es dort ein High-Char-Register AH gibt, dass am selben Ort startet wie das Short-Size-Register AX (siehe Abbildung 11). Da wir

keine zufriedenstellende Lösung finden konnten, entschieden wir uns die Register nicht speziell zu beschriften, sondern die Animationen so zu gestalten, dass bei einem Zugriff auf ein Unterregister die entsprechenden Bytes im Long-Size-Register animiert werden.

5.13 Animationen überspringen

Wir haben uns in den Anforderungen überlegt, dass Personen mit guten Vorkenntnissen schnell gelangweilt sind, wenn sie gewisse Animationen, die sie bereits verstanden haben, immer wieder anschauen müssen. Wir haben auch während der Entwicklung bei den Systemtests gemerkt, dass es mühsam ist, immer alle Schritte durchzuklicken, wenn man nur etwas Bestimmtes testen möchte.

Unsere erste Idee war, dass man ganze Instruktionen überspringen kann. Wir haben aber bemerkt, dass es einen grösseren Nutzen bringt, wenn man die Schritte “Get Instruction” und “Increment Instruction Pointer” einzeln ausschalten kann. Diese begreift man wahrscheinlich schneller, da sie immer gleich ablaufen. Der spannendere und abwechslungsreichere Schritt ist “Execute Instruction”. Der Nachteil dieser Methode ist, dass man, wenn man alle Schritte ausgeschaltet hat, trotzdem drei Mal klicken muss, um zur nächsten Instruktion zu kommen. Es wäre bei diesem Ansatz kein Problem in einer Folgearbeit einen zusätzlichen Button hinzuzufügen, der den ganzen Schritt überspringt. Deshalb haben wir uns dazu entschieden die Animationen schrittweise ausschaltbar zu machen.

5.14 Change Log

Eine Anforderung an uns war, dass die historischen Zustände nach jedem durchgeführten Schritt tabellarisch dargestellt werden. Wir wollten diese in einem Change Log anzeigen, das man ein- und ausblenden kann. Da wir die Änderungen der Schritte “Get Instruction” und “Increment Instruction Pointer” zu wenig informativ fanden, wollten wir nach der Ausführung der Instruktion jeweils einen neuen Eintrag hinzufügen. Nun stellte sich die Frage, welche Daten man in diesem Log anzeigt. Die Assembly Interpretation des ausgeführten Maschinencodes ist sehr wichtig, damit man nachvollziehen kann, zu welcher Instruktion die Änderungen gehören. Zudem wollten wir nicht nur die Werte der Operanden anzeigen, da man bei einem “push” nicht sehen würde, dass das Register RSP angepasst wird. Um die Informationsüberlastung zu reduzieren, entschieden wir uns nur die Schreibzugriffe darzustellen. Der Inhalt passte somit auch zum Namen Change Log.

Nachdem Entscheid, was im Change Log angezeigt werden soll, haben wir in Microsoft PowerPoint das Design entwickelt, das wir umsetzen wollten (siehe Abbildung 12). Mit Hilfe eines Buttons lässt man das Change Log auf der rechten Seite einblenden und kann es dann mit einem anderen Button wieder ausblenden. Dieses Design wollten wir im ersten Usability Test auf die Verständlichkeit überprüfen.

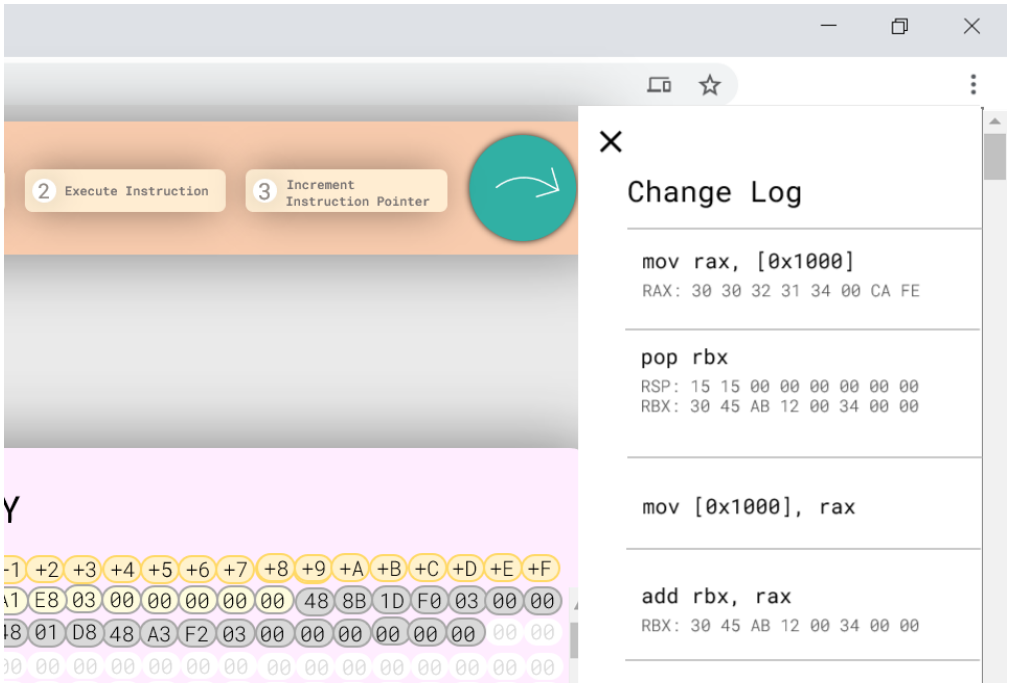


Abbildung 12 Designansatz Change Log

6 Umsetzung

In diesem Kapitel beschreiben wir, wie wir unser Lösungskonzept umgesetzt haben, welche Probleme dabei aufgetreten sind und wie wir diese gelöst haben. Im Anhang C befinden sich zusätzliche Informationen, die bei der Weiterentwicklung des Codes hilfreich sind. Dort werden zum Beispiel die Architekturkomponenten genauer beschrieben und Diagramme angezeigt. Im Hauptteil werden Namen der Elemente aus dem Code für eine bessere Leserlichkeit jeweils **fett** formatiert dargestellt.

6.1 Generische Animation der Instruktionen

Für die generische Animation der Instruktion benötigten wir Informationen über die Lese- und Schreibzugriffe der Instruktionen. Die Capstone Library in C banden wir in der Studienarbeit nicht direkt ein, sondern verwendeten den Wrapper Capstone.js[9], der dies für uns erledigt. In Capstone.js erhalten wir jedoch nur generelle Informationen über die Instruktion. Die gewünschten detaillierten Informationen müssen wir nun selbst aus Capstone auslesen.

6.1.1 Instruktionsdetails aus Capstone

Da wir uns bereits in der Studienarbeit mit der API und den Files im originalen Capstone Repository auseinandergesetzt hatten, wussten wir, dass es viele Beispielprogramme gibt, die zeigen, wie man Capstone verwendet. Wir schauten uns erneut einige Files an und entdeckten ein Unittest File, das die ganzen Instruktionsdetails auslas und schön formatiert ausgab [6]. Es handelte sich dabei um circa 370 Zeilen C Code und wir waren froh, dass wir uns den Zeitaufwand, diese selbst zu implementieren sparen konnten. Wir fügten diesen Code in das Capstone "cs.c" File ein, so dass wir die Funktion im Capstone.js Wrapper aufrufen konnten. Danach konnten wir die Details der Instruktionen ausgeben. Leider werden diese in der Funktion jedoch mit "printf" gedruckt und erscheinen somit in der Browser Debugger Konsole. Wir konnten keinen Weg finden, um den Output-Stream im TypeScript abzuspeichern. So blieb uns nichts anderes übrig als die C Funktion umzuschreiben, so dass sie einen grossen Buffer entgegennimmt, die Instruktionsdetails darin abfüllt und diesen nach dem Aufruf zurückgibt. Mit Hilfe von “sprintf” schrieben wir jeweils die Werte in einen temporären Buffer und hängten diesen mit "strcat" an den Output Buffer an, um die Details

[9] Capstone.js: https://alexaltea.github.io/capstone.js/

schliesslich im Capstone.js Wrapper als String auszulesen. Um die Daten im TypeScript direkt in ein Objekt abfüllen zu können, änderten wir die Ausgabe im C Code so ab, dass sie die Daten im JSON Format in den Buffer schreibt.

6.1.2 InstructionOperands Objekt

Von Capstone erhalten wir nun die Informationen zur Instruktion und zu den Operanden. Dort werden auch Zugriffe auf die Register und Flags erkannt, die nicht in den Operanden der Instruktionen vorkommen, zum Beispiel das RSP Register bei “push”. Wir wissen so zwar nicht, was in die Register geschrieben wird, wir können diese jedoch vor und nach der Ausführung einer Instruktion auslesen und so die Werte bestimmen.

Beim Memory Zugriff erhalten wir von Capstone nur Informationen, wenn es sich dabei um einen Operanden handelt. Zudem erhalten wir die Adresse nicht, auf die zugegriffen wird. Deshalb nutzen wir dort einen Memory Access Hook, den Unicorn anbietet. Wir erhalten in der Callback-Funktion die Memory Adresse, auf die zugegriffen wurde und wie viele Bytes betroffen sind. Bei einem Schreibzugriff erhalten wir zusätzlich den Wert, der geschrieben wird. Beim Read können wir mit den gegebenen Informationen auf das Memory zugreifen und den Wert auslesen.

Der von Capstone erhaltene String im JSON Format übergeben wir dem **instructionOperandsService**, der ein **InstructionOperands** Objekt (siehe Abbildung 15) erstellt, dass wir dem **Instruction** Objekt hinzufügen. Abbildung 13 und Abbildung 14 zeigen Ausschnitte aus dem String, den wir von Capstone für die jeweiligen Instruktionen erhalten. Bei den Operanden gibt es drei Typen “MEM” (Memory), “IMM” (Immediate) und “REG” (Register). Die Immediate ist eine Art Konstante, die man bei gewissen Instruktionen als Operand angeben kann.

```
"op_count": "1",
"operands": [
    {"type": "MEM", "reg_index": "rax", "scale": "8", "size": "8", "access": "READ" }],
"registers_read": [ "rsp", "rax"],
"registers_modified": [ "rsp"]
```

*Abbildung 13 Details von Capstone zur Instruktion: "push qword [rax * 8]"*

```
"op_count": "1",
"operands": [
    {"type": "IMM", "value": "0xc", "size": "8" }],
"registers_read": [ "rflags"],
"EFLAGS": [
    { "name": "ZF", "access": "TEST" }]
```

Abbildung 14 Details von Capstone zur Instruktion "jnz 0xc"

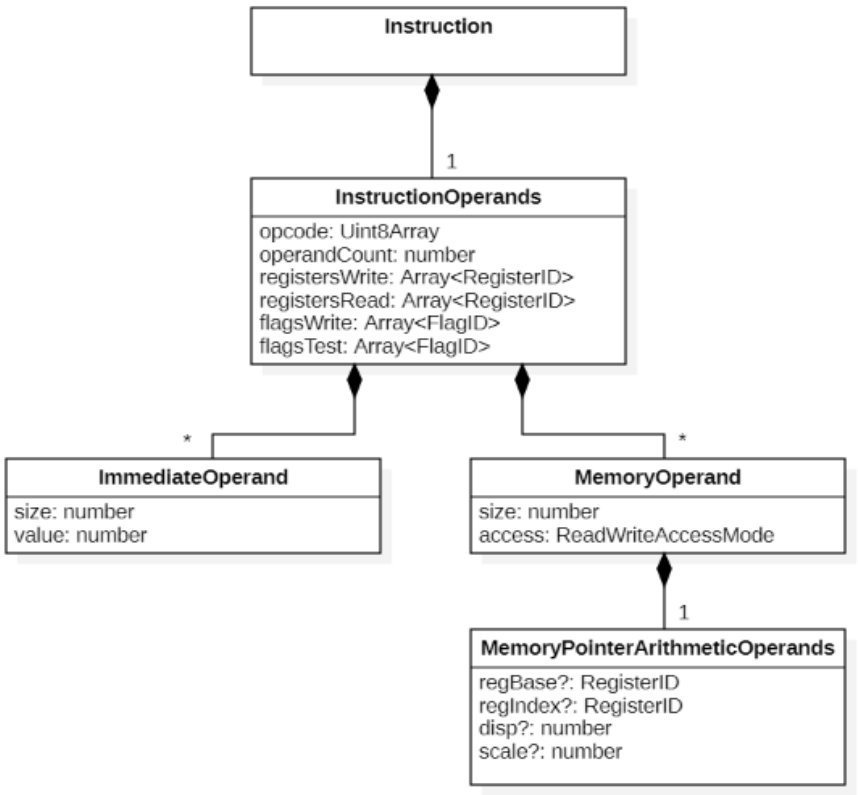


Abbildung 15 InstructionOperands Objekt

Bei den Register Operanden fügen wir die entsprechende **RegisterID** je nach Zugriffsmodus dem **RegistersRead** und/oder dem **RegisterWrite** Array in den Attributen des **InstructionOperands** hinzu. Die Flags füllen wir ähnlich wie die Register ab. Wir erhalten von Capstone einen Array mit den Flags und füllen diese je nach Zugriffsmodus in die **FlagsWrite** oder **FlagsTest** Array im **InstructionOperands** Objekt ab.

6.1.3 AccessedElements Objekt

Im **InstructionOperands** Objekt erhalten wir nur den Namen und die Adressen der Elemente, auf die die CPU während der Ausführung zugreift. Für die Animationen der betroffenen Elemente, benötigen wir die richtigen Inhalte, die gelesen und geschrieben werden.

Deshalb haben wir dem **State** ein Attribut mit einem **AccessedElements** Objekt (siehe Abbildung 16) hinzugefügt. Im Schritt “Get Instruction” wird im **Controller** bereits die Instruktion ausgeführt, die dann im Schritt “Execute Instruction” animiert werden soll. Wenn während der Ausführung der Instruktion im Emulator auf das Memory zugegriffen wird, wird der Memory Access Hook aktiviert. Anhand der Informationen aus der Callback Funktion können wir die Daten aus dem Memory auslesen und dem **AccessedElements** Objekt hinzufügen, um diese später im Schritt “Execute Instruction” zu animieren.

Vor und nach der Ausführung der Instruktion im Emulator lesen wir die Werte aus dem Emulator, die vom **InstructionOperands** Objekt vorgegeben sind und füllen diese in das **AccessedElements** Objekt ab. Zum Beispiel werden die Werte derjenigen Register, deren **RegisterID** als Lesezugriffe im **InstructionOperand** gespeichert sind, als Register Objekte im **AccessedElements** Objekt als **registerReadAccess** gespeichert. Wenn die Bytes später animiert werden, kann der **AnimationService**, anhand der **LocationId** der Bytes in den **Register** Objekten erkennen, welches Byte im HTML geklont und animiert werden soll.

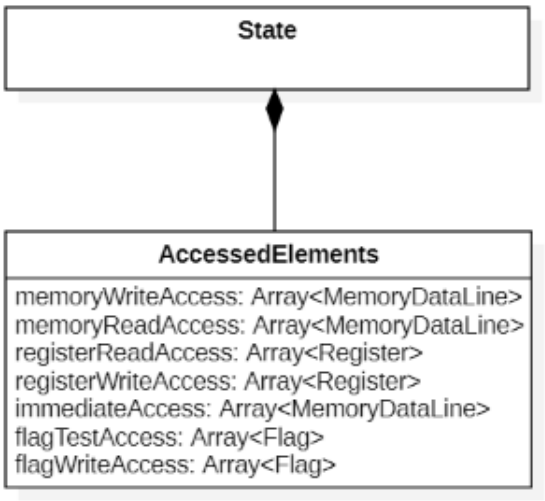


Abbildung 16 AccessedElements Objekt

6.1.4 Ablauf generische Animation der Instruktion

Die Schwierigkeit war nun, in welcher Reihenfolge Daten gelesen und geschrieben werden sollen. Wir haben uns dazu die Instruktionen angeschaut, die in der Vorlesung Bsys1 verwendet werden und im Anhang B aufgeführt sind. Bei den meisten werden wie bereits ver-

mutet zuerst Werte gelesen und danach wieder geschrieben. Es gibt jedoch auch Funktionen, wie “push” bei denen zuerst der Stack-Pointer gelesen und geschrieben wird und danach der gewünschte Wert ausgelesen und in den Stack kopiert wird. Da dies jedoch die Ausnahme ist und für die meisten anderen Instruktionen unser Ansatz passte, haben wir uns für den folgenden Ablauf entschieden:

1. Flags lesen
2. Register lesen
3. Memory lesen
4. Immediates lesen
5. Werte in Memory schreiben
6. Werte in Register schreiben
7. Werte in Flags schreiben

In der folgenden Tabelle 1 wird die Reihenfolge an ein paar Beispiel Instruktionen gezeigt.

add ebx, [rbp + 0x10]	**push qword [rax * 8]**	**jnz 0xc**
1. EBX und RBP lesen 2. Memory ab [rbp + 0x10] lesen 3. EBX schreiben 4. Flags schreiben	1. RSP und RAX lesen 2. Memory ab [rax * 8] lesen 3. Wert auf Stack im Memory schreiben 4. RSP schreiben	1. Flags lesen 2. Immediate 0xc lesen

Tabelle 1 Beispiele für den Ablauf der generischen Animation

Um diese Abfolge im Schritt “Execute Instruction” zu zeigen, führen wir die generische Instruktion im **animationController** aus. Diese führt zuerst nacheinander die Lesezugriffe und den Immediate Zugriff aus. Durch die **accessedElements** weiss der **animationService** genau, welche Elemente geklont werden müssen und dann zu ihrem Platz in der CPU fliegen. Danach werden die Schreibzugriffe animiert, wo die **accessedElements** eingeblendet werden und dann zu ihrem Ziel in den Registern oder dem Memory fliegen. Die **accessedElements** wie zum Beispiel Register werden jeweils nacheinander einzeln animiert. Die Bytes des jeweiligen Elements fliegen gemeinsam an ihr Ziel.

Unter den Instruktionen, die in Bsys1 verwendet werden, gibt es einige arithmetischen Funktionen, bei denen alle Flags modifiziert werden. Da es sehr lange dauert, bis alle vier Flags einzeln an ihr Ziel fliegen, würden die arithmetischen Funktionen schnell langweilig werden.

Deshalb haben wir uns entschieden, die von einem Lese- oder Schreibzugriff betroffenen Flags gemeinsam zu animieren und nicht alle einzeln.

6.1.5 Attention Animation

Um die Verständlichkeit der Animationen zu verbessern haben wir die Puls-Animation aus dem GUI-Designansatz der Studienarbeit umgesetzt. Diese lässt die Elemente, die in die Ausführung eines Schritts des Prozessor-Zyklus involviert sind, kurz pulsieren. Das bedeutet, dass ein grösser werdender Kreis rund um das Element dargestellt wird. Zum Beispiel gibt es vor der Bewegung der Bytes einen Puls Effekt auf die betroffenen Bytes und das Zielelement. Anhand der HTML-Id des Elements, kann man ihm eine CSS-Klasse hinzufügen, die den Animationseffekt startet.

Bei der Umsetzung merkten wir, dass bei der Bewegung der Bytes mehr nötig ist, um die Aufmerksamkeit des Benutzenden auf sich zu ziehen. Dieser soll die Animation auch im peripheren Sehen wahrnehmen. Da die einzelnen Bytes nur einen sehr kleinen Teil des Bildschirms einnehmen, mussten wir ihre Animation anpassen, damit sie besser sichtbar werden. Sie werden nun in der Animation um 50% vergrössert und leicht gedreht. Ausserdem wurde der Puls um das Element verstärkt. Der Ablauf der Animation im GUI ist in der Abbildung 17 ersichtlich. Der Benutzende wird, sobald er die Animation wahrgenommen hat, auf die betroffenen Bytes blicken und kann somit die Bewegungsanimation von Anfang an beobachten. Leider dauern die Animationen nun sehr lange. Wir lassen sie bewusst nicht schneller abspielen, um Personen, die den Simulator zum ersten Mal benutzen nicht zu überfordern. Man könnte in einer Folgearbeit komplexere Animationen entwickeln, die schneller ablaufen und trotzdem gut erkannt werden.

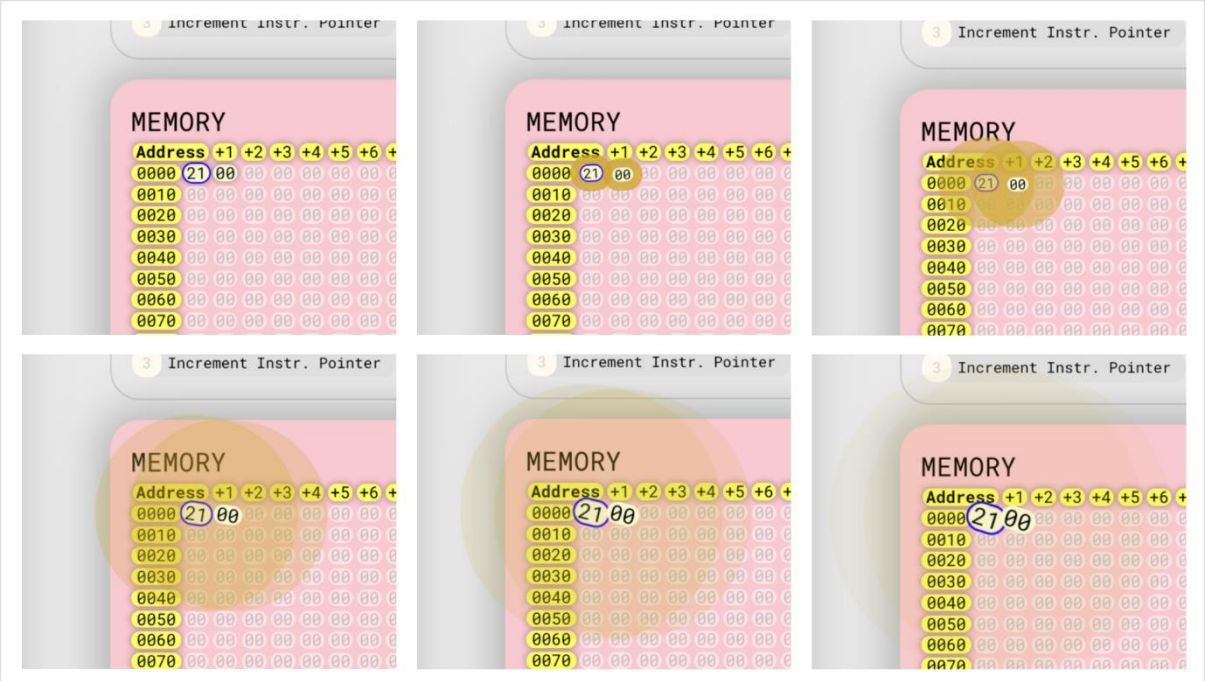


Abbildung 17 Ablauf der Attention Animation

6.1.6 Verständlichkeit der generischen Animation

Wir haben diese Animationen im ersten Usability Test getestet. Ein Teilnehmer, der Wirtschaft studiert, war trotz der fehlenden Informatikkenntnissen in der Lage, die Funktionsweise gewisser Assembly Instruktionen zu erkennen. Unser Ansatz funktionierte also und stellte sich als die bessere Entscheidung heraus, als jede Instruktion einzeln zu animieren, wie es in der Studienarbeit geplant war, was nur ein kleines Subset an Instruktionen angeboten hätte.

Auch die anderen Teilnehmenden des ersten Usability Tests fanden die Animationen verständlich. Einzig die Jump Instruktion funktioniert in diesem Konzept nicht ganz. Diese überschreibt unter gewissen Bedingungen den Instruction Pointer, um an einen anderen Ort im Code zu springen, zum Beispiel um einen Loop auszuführen. Im Simulator wird dies so animiert, dass nur die Input-Werte eingelesen werden. Man sieht jedoch nicht, dass der Instruction-Pointer überschrieben wird. Beim Schritt “Increment Instruction Pointer” wird dann der Instruction Pointer fälschlicherweise, wie in den anderen Instruktionen um die Länge der Instruktion erhöht. Er zeigt danach jedoch die richtige Zieladresse an, die man in der Instruktion als Operanden mitgegeben hat. Somit müsste man diesen Animationsablauf noch anpassen, so dass ersichtlich ist, dass der Instruction-Pointer von der Jump Instruktion überschrieben wird.

Es gäbe hier zwei Möglichkeiten, das Problem zu lösen, entweder man fügt eine Animation während des "Execute Instruction" Schritt hinzu, die den Instruction Pointer überschreibt und überspringt danach den "Increment Instruction" Schritt. Oder man würde im "Increment Instruction" Schritt eine andere Animation abspielen, die den Instruction Pointer überschreibt. Wir haben uns jedoch aus Zeitgründen dagegen entschieden, eine dieser Lösungen umzusetzen. Erstens müsste man diese Ansätze in einem weiteren Usability Test auf ihre Verständlichkeit überprüfen, um den richtigen zu wählen. Und zweitens haben die Teilnehmenden trotzdem erkannt, dass der Instruction Pointer gesetzt wird und dass sie sich zum Beispiel in einem Loop befinden. Von dem her empfanden wir die Umsetzung nicht als notwendig, um die Verwendbarkeit des Produkts zu gewährleisten.

6.2 CPU-Box

Wir hatten uns entschieden, die Instruktionsverarbeitung wie ein schwarzes Loch zu gestalten, in dem die Input Daten verschwinden. Nach der Umsetzung stellten wir fest, dass diese Darstellung etwas seltsam wirkte. Beim Lesezugriff flogen die Daten zu einem unsichtbaren Ziel oberhalb der Current Instruction, legten sich übereinander und verschwanden plötzlich (siehe Abbildung 18). Man konnte weder erkennen, warum die Daten dorthin fliegen, noch sehen, dass die CPU diese Daten verarbeitet. Da kam uns die Idee der CPU-Box, die die Lesezugriffe als Input nimmt und die Schreibzugriffe als Output ausgibt.

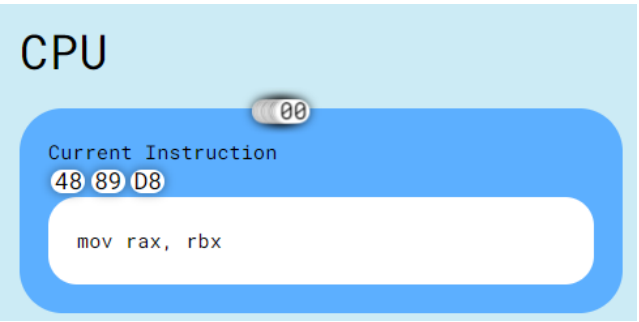


Abbildung 18 Erste Umsetzung des Lesezugriffs

6.2.1 Design CPU-Box

Die CPU-Box hat leere Platzhalter links von sich, zu denen die Daten der Lesezugriffe nacheinander hinfliegen. Danach fährt die Box nach links und verschlingt dabei die Inputs. Gleichzeitig erscheinen auf der rechten Seite die Outputs, die geschrieben werden sollen. Die Outputs fliegen anschliessend nacheinander an ihr Ziel und die Box bewegt sich wieder nach rechts in ihre Ausgangsposition. Der Ablauf ist in der Abbildung 19 ersichtlich.

Um sicherzugehen, dass wir auf dem richtigen Weg sind, haben wir unseren umgesetzten Ansatz mit dem schwarzen Loch im ersten Usability Test überprüft und unsere Vermutung, dass es verwirrend ist, wenn die Daten kein Ziel haben und einfach verschwinden, hat sich bestätigt. Wir zeigten den Testpersonen nach dem Test unseren Design-Ansatz für die CPU-Box, den wir in Microsoft PowerPoint animiert haben (siehe Abbildung 19). Sie haben direkt verstanden, dass die Darstellung die Verarbeitung der Daten darstellt.

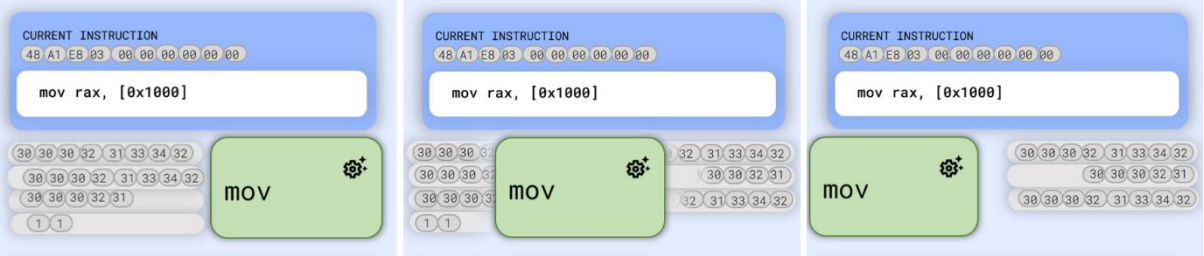


Abbildung 19 Designansatz CPU-Box

6.2.2 Umsetzung CPU-Box

Wir stellen die CPU-Box in der Vue-Komponente **ExecutionBox** dar, die die Informationen aus den **accessedElements** des **State** Objekts erhält. Damit man besser erkennt, welche Daten gelesen und geschrieben werden, haben wir Informationen über die Werte, wie die Registernamen oder Adressen auf die CPU-Box geschrieben. Als Instruktionsnamen in der Mitte verwenden wir jeweils das erste Wort der Assembly Interpretation der Instruktion.

Beim Start des Schritts “Execute Instruction” werden die Daten, die gelesen werden, animiert und zum Schluss jeweils im Platzhalter links von der CPU-Box eingeblendet. Wenn die CPU-Box nach dem Lesen der Input Elemente nach links fährt, kommen die Output Elemente zum Vorschein (siehe Abbildung 20). Diese fliegen nacheinander an ihr Ziel, wo nach ihrer Ankunft die Daten aktualisiert werden.

Abbildung 20 Umsetzung CPU-Box

6.3 Scrolling

Die Implementation des Scrolling an die richtige Position im Speicher war mithilfe der “ScrollIntoView” Funktion[10] mit nur ca. fünf Zeilen Code erledigt. Dieser Funktion kann man den Parameter “smooth” übergeben und damit die Positionsveränderung live mit verfolgen. Während der Implementation haben wir jedoch festgestellt, dass der “smooth” Parameter vom Safari Browser nicht unterstützt wird. Dies war nicht weiter schlimm, da wir uns in den nichtfunktionalen Anforderungen nur auf den Chrome Browser fokussiert haben.

Als wir jedoch damit begonnen haben, die komplexen Animationen der generischen Instruktionen umzusetzen, stellten wir fest, dass uns die “ScrollIntoView” Funktion keine Möglichkeit anbietet, zu erfahren, wann die Scroll-Animation fertig ist. Dies wurde im Webstandard nicht eingebaut und wartet als Verbesserungsvorschlag[11] seine Einführung ab. [7]
Es war uns also nicht möglich unsere Animationen richtig zu koordinieren, da wir nicht wussten, wann die Scroll-Animationen fertig waren und die Bewegung der Bytes starten sollte. Schlussendlich haben wir die “scroll-polyfill” Library[12] verwendet, die den Web-Standard um die fehlende Funktionalität erweitert. Dadurch erhielten wir von der Funktionen ein Promise zurück, den wir abwarten konnten, um danach die nächste Animation zu starten. Ein positiver Nebeneffekt dieser Library ist, dass das Smooth-Scrolling auch im Safari Browser funktioniert.

6.4 Einbau des Nasm und Ndisasm

Für die Kompilation von Unicorn und Capstone verwendete der Entwickler von Unicorn.js und Capstone.js Emscripten. Dabei handelt es sich um eine Build und Compiler Toolchain, die bestehende C und C++ Projekte für den Gebrauch im Web umwandelt. Dazu verwendet Emscripten LLVM und bietet ein Drop-in Replacement für den GCC Compiler. Aus diesem Grund wollten wir Emscripten für die Nasm Kompilation einsetzen. [8]

[10] ScrollIntoView Funktion: https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView
[11] GitHub Issue “Provide onAnimationEnd callback in scrollIntoView options” von Stanislav Lashmanov: https://github.com/w3c/csswg-drafts/issues/3744
[12] scroll-polyfill: https://www.npmjs.com/package/scroll-polyfill

6.4.1 Kompilation

Da Nasm ein Projekt ist, das von vielen verschiedenen Architekturen unterstützt wird, waren wir optimistisch, dass die Kompilation auf die geplante Weise gelingen wird. Zusätzlich hatten wir ein Webprojekt entdeckt, das den Nasm im WebAssembly Format verwendet [9]. Wir wussten also, dass die Portierung möglich sein sollte. Da wir bereits getane Arbeit nicht wiederholen wollten, nutzen wir die Anleitung zur Kompilierung des Nasm, die das Projekt zur Verfügung stellt [10].Leider reichte diese jedoch nicht für uns aus, da sie veraltet war. Wir stellten zudem fest, dass die Nasm Library nicht sauber, sondern über viele Hacks in das Projekt eingebunden wurde. Deshalb beendeten wir diesen Weg vorzeitig und setzten uns selbst mit der Emscripten Dokumentation auseinander.

Als erstes mussten wir ein paar Environment Variablen für Emscripten richtig setzen, bis uns die Kompilation schliesslich gelang. Jedoch lagen die daraus resultierenden Files im falschen Format vor. Um die Library in unser HTML-File integrieren zu können, muss der ganze Librarycode zu einem JavaScript ES6 Modul[13] in ein einzelnes JavaScript File zusammengefügt werden. Da uns die Zeit fehlte, um uns mit Make Buildfiles auseinanderzusetzen, suchten wir einen schnelleren Weg, um das passende File zu generieren.

Während der Kompilation gibt Emscripten die Befehle an den Compiler einzeln in der Konsole aus, bevor diese ausgeführt werden. Wir bemerkten, dass es ausreicht, denjenigen Befehl, der die Library konstruiert nach der Kompilation noch einmal mit den richtigen Parametern auszuführen. Um die richtigen Einstellungen vorzunehmen, orientierten wir uns an denjenigen, die Unicorn.js verwendet. So gelang es uns ohne grossen Zeitverlust, die Files im richtigen Format zu erhalten. Nasm hat uns mit der folgenden Fehlermeldung im Browser begrüsst: "nasm: fatal: no input file specified Type nasm -h for help". Diese signalisiert, dass Nasm nun bereit ist, ein Input File entgegenzunehmen. Danach konnten wir den Ndisasm auf dieselbe Weise kompilieren. Da die resultierenden Library Files jeweils sehr gross waren, haben wir mit Hilfe von den Compiler Flags die Compiler von schneller Performance auf kleine File Grösse umgestellt und stellten zusätzlich die Debug Informationen aus. So erreichten wir eine akzeptable Grösse.

[13] JavaScript Modul: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

6.4.2 Einbindung Nasm

Wir mussten nun eine Möglichkeit finden, dem Nasm ein Assembly File zu übergeben, das er dann in Maschinencode umwandeln kann und in einem anderen File zurückgibt. Die Schwierigkeit hierbei war, dass wir im Browser keinen Zugriff auf echte Files, sondern auf eine emulierte Dateisystem-API haben. Beim Lesen der Dokumentation von Emscripten bemerkten wir ausserdem, dass wir in den Technologien, die wir verwenden können, eingeschränkt sind, da unsere Auslieferung der HTML-Datei über das file://-Protokoll[14] erfolgt. Es stellte sich jedoch heraus, dass wir trotzdem eine der vier Dateisystem-APIs verwenden können, und zwar MEMFS die mit einer In-Memory-Methode arbeitet. [11]

Der Ablauf wie Emscripten die C Programme im Browser startet ist sehr komplex. Zuerst wird die ganze C Umgebung aufgebaut, dann automatisch die Main Methode ausgeführt und danach wird die Umgebung wieder abgebaut. Da wir keinen Zugriff auf das Filesystem hatten, bevor die Main Methode startete, war es nicht möglich, Nasm unsere In- und Output Files zu übergeben. Es erforderte viel “Trial and Error”, bis wir die Emscripten Umgebung beherrschten. Zusätzlich mussten wir Nasm neu kompilieren, um Zugriff auf alle relevanten Funktionen zu erhalten, da diese standardmässig nicht zum Gebrauch exportiert werden, um im File Platz zu sparen. Schlussendlich gelang es uns, das Nasm Modul bei der Initialisierung davon abzuhalten, automatisch die Main Methode zu starten. So können wir die Filesystem API nutzen, um das Assembly Input File und ein leeres Output File zu erstellen und danach die Main Methode des Nasm mit den richtigen Flags selbst aufzurufen. Danach können wir das Output File auslesen und dann die Modul Instanz wieder löschen.

Noch etwas unschön ist, das Nasm keine Konsolen Flags für die Aktivierung des 64-Bit Modus bereitstellt. Man kann den 64-Bit Modus nur aktivieren, wenn man im Assembly Code selbst "BITS 64" an den Anfang schreibt. Das Problem ist nun, dass man diesen Modus bei der Bearbeitung des Codes aus Versehen löschen kann. Man erhält dann entweder einen Fehler im Nasm oder im Emulator, da er einen 16-Bit Maschinencode erhält. Man könnte dieses Problem umgehen, indem man vor der Übergabe des Assembly Codes an Nasm, den Modus einfügt, ohne dass es im Editor ersichtlich ist. Oder man könnte den Nasm Source Code umschreiben, um immer im 64-Bit Modus zu laufen.

[14] file://-Protokoll: https://datatracker.ietf.org/doc/html/rfc8089

6.4.3 Einbindung Ndisasm

Als nächstes haben wir auch den Ndisasm portiert. Jedoch gibt Ndisasm den Output nicht in einem File aus, sondern schreibt ihn auf die Konsole, was in unserem Fall die Browser Debugger Konsole ist. Wir fanden nach langem Suchen heraus, wie man die Konsolenausgabe im TypeScript abfangen konnte, um den Inhalt in eine Variable zu speichern. Auf dieselbe Weise können wir auch die Fehlerausgabe von Nasm abfangen, um sie dann unterhalb des Editors dem Benutzenden anzuzeigen. Hätten wir diesen Mechanismus bereits gekannt, als wir die Ausgabe der Informationen über die Instruktionen im Capstone umsetzten, hätten wir uns viel C Programmierung ersparen können. Auf diese Weise hätten wir nämlich nicht den ganzen Print Mechanismus und den Funktionsaufruf umstellen müssen, sondern hätten direkt den Output abfangen können. Aber im Nachhinein ist man bekanntlich immer schlauer.

Als wir Ndisasm fertig implementiert hatten, merkten wir, dass die Ausgabe falsch ist und wir nicht dieselbe erhalten, wie wenn Ndisasm direkt auf dem Betriebssystem ausgeführt wird. Wir waren nicht sicher, ob wir bei der Kompilation einen Fehler gemacht hatten oder ob der Disassembler auf der Webumgebung nicht lauffähig ist, da er vielleicht schlechter unterhalten ist als der Nasm. Wir untersuchten daraufhin den Sourcecode des Ndisasm, in der Hoffnung dort einen Grund für das Fehlverhalten zu finden. Als der Erfolg nach einigen Stunden Aufwand noch immer ausblieb, schilderten wir das Problem Prof. Stefan Richter, der uns umgehend die Lösung lieferte. Der Fehler lag nicht im C Code oder der Kompilation sondern in unserem TypeScript Code. Die Funktion "writeFile" der Filesystem API ist eine überladene Funktion, die ihre Funktionsweise ändert, je nachdem welcher Datentyp ihr übergeben wird. Die Funktion akzeptiert als Input einen String oder ein ArrayBufferView, wobei der String UTF-8 codiert ist und die ArrayBufferView als Rohdaten verarbeitet werden. Bei Nasm übergaben wir einen String, was auch korrekt war, sowohl für den Assembly Code wie auch für das Output File. Bei der Einbindung des Ndisasm verwendeten wir denselben Code, wodurch der Disassembler den Maschinencode in einem falsch codierten Format erhielt und dementsprechend eine falsche Ausgabe erzeugte. Vor lauter komplexer Kompilation scheiterte es zum Schluss also an einem einfachen Funktionsaufruf. Nach der Korrektur waren beide Programme funktionsfähig und gaben dieselben Resultate aus, wie die Programme auf dem Betriebssystem.

Es hat eine ganze Weile gedauert, bis wir die Programme im Browser zum Laufen gebracht haben. Da wir viel Zeit für dieses wichtige Feature aufwenden mussten, hat man in unserem

Projekt während eines ganzen Sprint keinen visuellen Fortschritt gesehen. Dies führte zu etwas Verunsicherung seitens des Betreuers, der gleichzeitig auch unser Kunde ist, als wir in der Sitzung nichts Neues zeigen konnten und sich unser Projekt um eine Woche verspätete. Wir konnten jedoch den Mehrwert des Produktes für die Studierenden enorm steigern, da sie nun den Assembly-Code aus der Vorlesung sowohl auf dem Betriebssystem als auch im Simulator ausführen können. Somit hatte es sich gelohnt das Risiko einzugehen die C-Libraries nach JavaScript zu portieren.

6.4.4 Workaround Jump und Call Instruktionen

Beim Testen der verschiedenen Instruktionen, die in Bsys1 verwendet werden, bemerkten wir, dass bei den Assembly Übersetzungen der Jump und Call Instruktionen im Simulator sonderbare Adressen als Operanden angezeigt werden. Zum Beispiel wird bei der Instruktion “call 0x7b” der Instruction Pointer korrekt zur Adresse 0x7b verschoben, die im Simulator angezeigte Assembly Übersetzung von Ndisasm zeigt jedoch “call 0x76” an. Es stellte sich heraus, dass der Ndisasm nur die richtige Adresse anzeigt, wenn die Instruktion an ihrer richtigen Position im Assembly Programm liegt. Wenn wir also nicht den ganzen Programmcode, sondern jede Instruktion einzeln übersetzen, erhalten wir jeweils falsche Adressen.

Durch Ausprobieren haben wir bemerkt, dass man zur Adresse, die Ndisasm ausgibt, noch die Adresse der aktuellen Instruktion dazu rechnen müsste. Im vorherigen Beispiel lag die Instruktion “call 0x7b” an der Adresse 0x5. Addiert man 0x5 zu 0x76 erhält man die Adresse 0x7b. Um dies im Code umzusetzen, muss man erkennen, ob es sich um eine Jump oder Call Instruktion handelt. Dann könnte man zur Adresse in der Immediate die Adresse der Instruktion addieren. Wir haben anhand der Opcodes, die wir von Capstone erhalten, versucht, Jump und Call zu identifizieren. Jedoch reicht diese Information allein nicht aus, um andere Instruktionen auszuschliessen. Zum Beispiel entspricht der Opcode 0xFF zwar einer Jump oder Call Instruktion, es kann sich jedoch auch um eine andere Operation wie Push handeln. Um die genaue Unterscheidung zu machen, müsste man wahrscheinlich die Opcode Erweiterung beachten, die in den Bits drei bis fünf des ModR/M Byte gesetzt werden. Call und Jump mit Opcode 0xFF haben die Bits (0b010 - 0b101) und Push 0b110 gesetzt. [12]

Leider fehlte uns die Zeit, um dieses Problem genauer zu untersuchen. Deshalb überprüften wir im **InstructionService**, ob es sich beim Opcode um eine Jump oder Call Instruktion handeln könnte und nahmen in diesem Fall die Assembly Übersetzung von Capstone anstatt derjenigen von Ndisasm. Dies hatte den Vorteil, dass wir bei allen anderen Instruktionen trotzdem die Nasm Syntax anzeigen konnten. Es brachte jedoch den Nachteil mit sich, dass wir dann teilweise auch bei Push Instruktionen die Capstone Syntax anzeigten, die nicht derjenigen aus der Vorlesung entsprach. Die Information, bei welchen Opcodes es sich um Call und Jump Instruktionen handelt, stammte von einer Webseite von Karel Lejska. Dieser hat die Opcodes aus verschiedenen offiziellen Quellen, wie zum Beispiel die Intel Manuals zusammengetragen. [13]

6.5 Code Editor

Ein weiterer wichtiger Teil, der in unserer Studienarbeit fehlte, war der Code-Editor. Wir wollten damit den Studierenden ermöglichen, ihre eigenen Programme auszuführen, um zu verstehen, was ihr Code macht. Zuerst wollten wir einen einfachen Texteditor verwenden, der in der von uns verwendeten Quasar-GUI-Library verfügbar war. Aber es wurde schnell klar, dass dieser für die Eingabe von Code ungeeignet war, da beim Drücken der Tab-Taste auf der Tastatur der Fokus innerhalb der Webseite zu einem anderen Element wechselte, anstatt den Text einzurücken, wie es beim Programmieren erwartet wird. Also mussten wir einen Weg finden, dieses Problem zu umgehen. Wir stiessen auf einen JavaScript-Code-Editor, der alle unsere Probleme löste und sich einfach einbinden liess. In diesem Editor ist die Bearbeitung von Code mit dem korrekten Tastatur Verhalten möglich und er bietet sogar ein Syntax-Highlighting für den Nasm an.

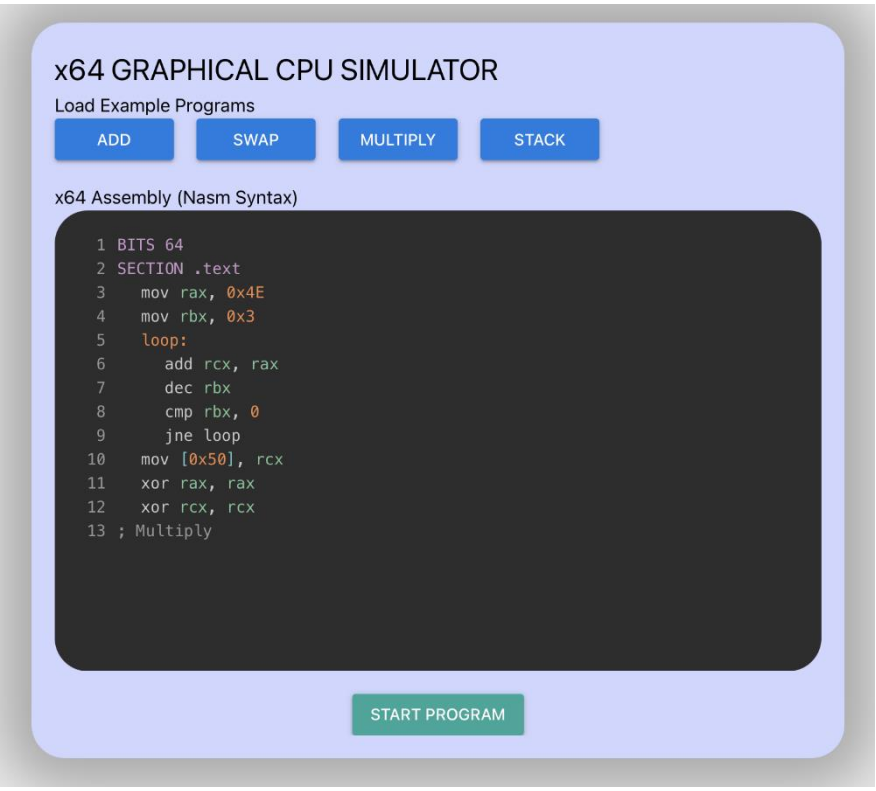


Abbildung 21 Umsetzung des Code Editors

Der Code Editor (siehe Abbildung 21) wurde als eigene Vue-Komponente entwickelt und wird als eigene Seite dargestellt. Dabei machten wir Gebrauch vom Vue-Router[15], der die verschiedenen Seiten, also den Editor und den Simulator, in der URL als Fragment abbilden kann. Das Fragment beginnt mit "#" und dahinter wird der Pfad simuliert. Zum Beispiel sieht der Aufbau der URL des Editors folgendermassen aus:

file:///Folder/index.html#/editor/

Da es sich beim Simulator und beim Code Editor um separate Seiten handelt, müssen Daten ausgetauscht werden. Der Editor muss zum Beispiel den Maschinencode an den Simulator liefern, sodass dieser überhaupt starten kann. Wir haben uns entschieden, die Daten über die URL zu übergeben.

[15] Vue Router: https://router.vuejs.org/

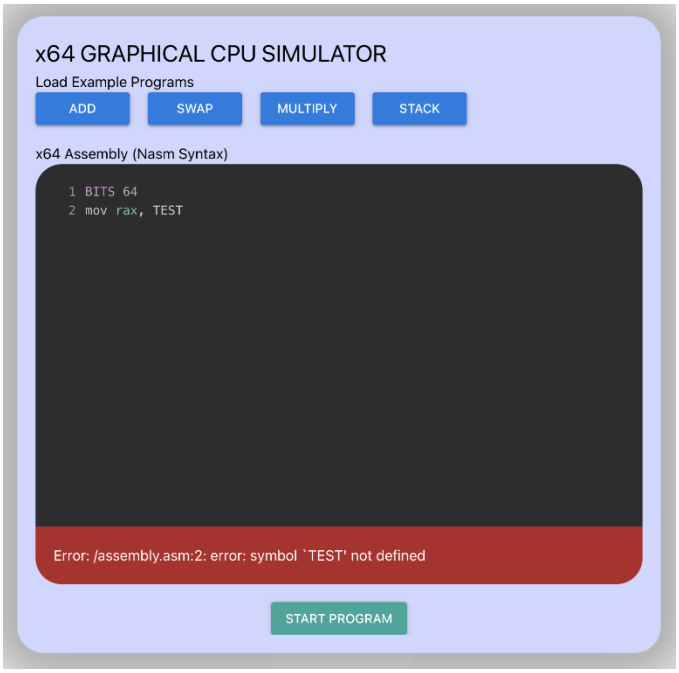


Abbildung 22 Fehlermeldung Code Editor

Mit einem Klick auf den “Start Program” Button wird der aktuell eingegeben Assembly Code an den Nasm übergeben und in Maschinencode umgewandelt. Tritt bei der Kompilation ein Fehler auf, wird dieser gleich unter dem Editor angezeigt (siehe Abbildung 22). Falls kein Fehler aufgetreten ist, wird der von Nasm produzierte Maschinencode als Hex-Zeichenkette in der URL übergeben. Zusätzlich wird der eingegebene Assembly Code Base64-kodiert und auch an die URL angehängt. So wird zum Beispiel aus dem untenstehenden Programm in der Abbildung 23 die URL in der Abbildung 24 erstellt:

```
1 BITS 64
2 mov rax, [0x0]
```

Abbildung 23 URL Beispiel - Programm

```
file:///Folder/index.html#/simulator/48,8B,04,25,00,00,00,00/QklUUyA2NAptb3YgcmF4LCBbMHgwXQ==
```

Abbildung 24 URL Beispiel - Link

Für die Übergabe des Maschinencodes wählten wir bewusst die Hex-Zeichenkette, da man so im Simulator erkennen kann, dass es sich in der URL um dieselben Daten handelt, die im

Memory dargestellt sind. Die Übergabe des Base64-kodierten Assembly Codes wird benötigt, damit der Simulator den originalen Assembly Code wieder an den Editor zurückgeben kann, falls man seinen Code editieren möchte. Dies ist nötig, da weder der Editor noch der Simulator einen State über die Seitennavigation hinweg speichern können.

Diese Lösung hat viele interessante Eigenschaften und Vorteile. Da unser HTML-File keinen Speicher hat, ist es dennoch möglich in der URL einen State zu speichern. Somit kann man die Seite neu laden und behält denselben Code bei. Man kann die Seite samt Programm als Bookmark im Browser speichern. Man kann sogar einer Kommilitonin oder einem Kommilitonen einen Teil der URL schicken, um zusammen zu arbeiten.

6.6 Beispielprogramme

Da der Simulator auch von Personen mit wenig Vorkenntnissen verwendet wird und diese sich mit der Assembly Programmierung vielleicht noch nicht so gut auskennen, bieten wir Beispielprogramme an. Es stellte sich uns jedoch die Frage, welche Programme dazu geeignet sind. Für den ersten Usability Test haben wir zwei Programme aus den Übungen der Bsys1 Vorlesung ausgesucht. Eines addierte zwei grosse Zahlen und das andere vertauschte zwei Variablen Werte im Memory. Um einen Loop anzubieten, schrieben wir ein Multiplikationsprogramm, das einen Multiplikand so oft addiert, wie der Multiplikator gross ist.

Während des Tests fiel uns jedoch auf, dass das Additionsprogramm viel zu lange dauerte und für Personen, die unserer Produkt kurz ausprobieren möchten, zu langweilig ist. Ausserdem fiel uns bei den Programmen aus der Vorlesung auf, dass diese verwirrend sind, wenn man den Stack nicht kennt oder noch nicht versteht. Bei einem Funktionsaufruf gibt es einen Prolog, bei dem man jeweils den Wert des Base Pointers auf den Stack legt (“push RBP”) und den Base Pointer auf den aktuellen Wert des Stacks setzt (siehe Abbildung 25). So kann man unter anderem auf die Argumente zugreifen, die man vor dem Funktionsaufruf auf dem Stack platziert hat. Die Testpersonen wussten dies jedoch nicht mehr und erkannten das Register RBP nicht als Base-Pointer Register. [5]

```
SECTION .text
global main
main:
  push rbp
  mov rbp, rsp
```

Abbildung 25 Prolog eines Funktionsaufrufs

Wir haben daraus die Erkenntnis gezogen, dass die Personen, wenn sie den Titel des Programms lesen, eine Erwartung haben, was passieren soll und die Nutzung des Stacks nicht dieser Erwartung entspricht. Ausserdem wird der Stack erst später in der Vorlesung erklärt und der Simulator sollte schon vorher einsetzbar sein. Deshalb haben wir dieses Stack Management aus den Funktionen entfernt und dafür ein eigenes Programm “Stack” hinzugefügt, das “push” und “pop” beinhaltet.

6.7 ByteInformation Objekt

Das **ByteInformation** Objekt ist ein Attribute des **State** Objekts. Es dient dazu, die Information ans GUI weiterzugeben, welche Bytes benutzt wurden und eingeblendet werden sollen und welche Bytes zum Stack-, Base- oder Instruction-Pointer gehören. Das Objekt besitzt einen Array **usedBytes**, in dem wir anfänglich alle **LocationIds** der Bytes abfüllten, die als benutzt dargestellt werden sollen. Dies führte jedoch bei grossen Programmen zu einem Performance Problem, da alle Bytes des Maschinencodes in den Array abgefüllt werden mussten und dieser somit riesig wurde. Da jede **ByteVue** Komponente im GUI, also insgesamt mehr als 4000 Elemente, jeweils ihre **LocationId** mit denjenigen des **usedBytes** Array abgleichen muss, führte unser Ansatz zu einem hohen Berechnungsaufwand.

Um das Problem zu beheben haben wir dem **ByteInformation** Objekt ein Attribut **CodeInformation** hinzugefügt, das die Startadresse und die Grösse des Codes angibt. So kann die **ByteVue** Komponente anhand der **LocationId** mit zwei Vergleichen entscheiden, ob sie im Codebereich liegt oder nicht. Da die Bytes aus dem Codebereich nicht mehr zu den **usedBytes** hinzugefügt werden müssen und dieser somit kleiner wurde, war die Performance wieder im selben Bereich, wie vor der Einführung des **ByteInformation** Objekts.

6.8 Darstellung der Pointer im Memory

Wir wollten unser Konzept für die Darstellung des Instruction-, Stack- und Base-Pointers im Simulator umsetzen. Zuerst mussten wir jedoch dafür sorgen, dass Programme, die den Stack verwenden, überhaupt im Simulator ausführbar sind. Bei einem “push” erhielten wir jeweils einen Fehler vom Emulator, dass er auf eine ungültige Speicheradresse zugreifen will. Grund dafür war, dass der Stack-Pointer im Emulator standardmässig auf null gesetzt ist und somit bei “push” auf eine negative Speicheradresse zeigt.

Da uns die Zeit fehlte, einen Fehlerdialog zu implementieren, wird bei Fehlermeldungen die Simulation abgebrochen und die Meldung in der Browser Konsole ausgegeben. Da sich die Benutzenden nicht gewöhnt sind, dass sie den Stack-Pointer vor der Verwendung auf einen anderen Wert setzen müssen, könnte dies verwirrend sein. Um dieses Problem zu umgehen, haben wir den Standardwert des Stack- und Base-Pointer an eine höhere Speicherposition gesetzt, so dass der Stack wachsen kann, ohne einen Fehler zu verursachen.

Andere Fehler, die vom Emulator geworfen werden, landen so natürlich immer noch in der Browser Konsole. Fehlerquellen sind zum Beispiel Zugriffe auf nicht adressierbares Memory oder Instruktionen, die der Emulator nicht unterstützt. Der Benutzende merkt in diesem Fall zwar anhand vom "Next Step" Button, der frühzeitig zu einem "Reload" Button wird, dass etwas schiefgelaufen ist, aber sieht die Fehlermeldung nicht. In einer Folgearbeit wäre es sicherlich hilfreich, die Fehler dem Benutzenden übersichtlich in einem Fehlerdialog anzuzeigen.

6.8.1 Stack- und Base-Pointer

Nachdem der Stack ohne Fehlermeldung nutzbar war, setzten wir die Darstellung der beiden Pointer im Simulator um. Dabei sind wir iterativ vorgegangen. Als ersten Schritt haben wir mithilfe des **ByteInformation** Objekts die Bytes, die zum Stack- oder Base-Pointer gehören eingefärbt. Zudem blendeten wir eine Pfeilspitze bei den Bytes ein, auf die die Pointer zeigen (siehe Abbildung 26). Die ursprüngliche Idee, die Pointer mit einem farbigen Rahmen darzustellen, haben wir verworfen, da wir, wenn der Stack- und Base-Pointer auf dasselbe Element zeigen, alle Indikatoren zusammen anzeigen möchten. Damit man einen Zusammenhang zwischen den Pointern und den Stack- und Base-Pointer-Registern RSP und RBP sehen kann, fügten wir diese zu den angezeigten Register hinzu (siehe Abbildung 27).

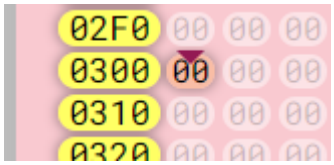


Abbildung 26 Erste Umsetzung des Stack-Pointers im Memory

Abbildung 27 Ausschnitte der ersten Umsetzung des Stacks

Danach haben wir diesen Ansatz im ersten Usability Test getestet. Die Testpersonen erkannten jedoch nicht, dass der Stack dargestellt wird. Sie haben zwar nach einer gewissen Zeit den Zusammenhang zwischen den Zahlen im Register RSP und der Position des roten Pfeils gesehen. Sie wussten jedoch nicht, dass es sich dabei um das Stack-Pointer-Register handelt. Nach dem Test haben sie sich gewünscht, dass man Tooltips anbietet für die Register und den zugehörigen Bytes im Memory.

Als wir unser ursprünglich geplanten Designansatz mit den eigenen Komponenten für die Pointer umsetzen wollten (siehe Abbildung 28 links), haben wir bemerkt, dass mit dieser Abstraktion die Erkenntnis verloren geht, dass es sich bei den Pointern in Wirklichkeit um ein Register handelt. Obwohl das in Bsys1 unterrichtet wird, wussten beide Testpersonen, die das Modul besucht hatten, nicht was RSP bedeuten soll. Wir haben uns deshalb entschieden die Pointer weiterhin bei den Registern anzuzeigen. Um jedoch zu verdeutlichen, dass es sich um eine Adresse handelt, zeigen wir ein gelbes Adressfeld neben den Registern an. Ausserdem weist ein Tooltip bei den beiden Registern darauf hin, dass es sich jeweils um das Stack-/Base-Pointer-Register handelt (siehe Abbildung 28 rechts).

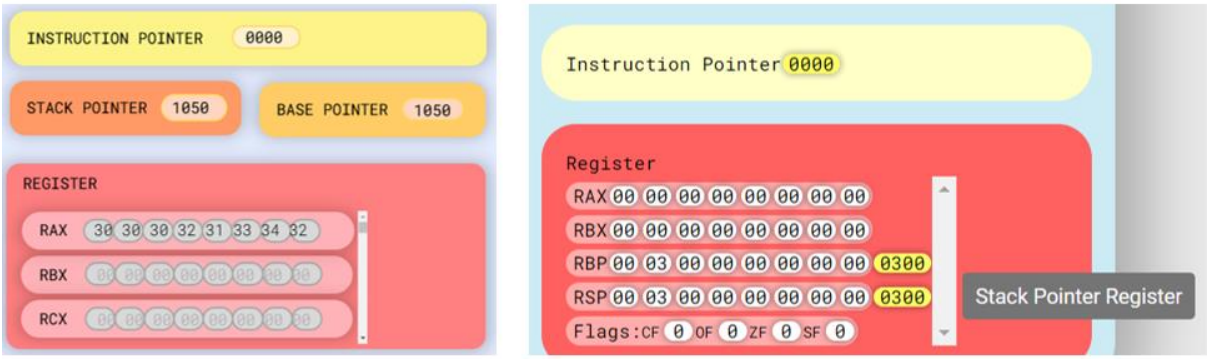


Abbildung 28 Gegenüberstellung Designansatzes und Umsetzung der Pointer in der CPU

Bei den Memory Bytes haben wir nach dem ersten Test ebenfalls Verbesserungen vorgenommen. Sobald der Pointer neu gesetzt wird, pulsieren die Pfeilspitzen kurz. Des Weiteren haben wir Tooltips bei den Bytes, auf die ein Pointer zeigt, hinzugefügt (siehe Abbildung 29). Wir haben darauf geachtet, dass die Tooltips auch an den Rändern des Memorys und in unvorteilhaften Konstellationen gut dargestellt werden. Da die Tooltips oben abgeschnitten würden, wenn sich das betroffene Byte in der ersten ersichtlichen Memoryzeile befindet, zeigen wir den Text nicht oberhalb des Bytes, sondern verschieben ihn zusammen mit der Pfeilspitze auf die linke Seite. Würde man den Text auf der rechten Seite anzeigen, würde er, wenn sich das Byte am Zeilenende befindet, über die Komponente ragen und abgeschnitten werden. Auf der linken Seite des Memorys existiert dieses Problem nicht, da sich dort die Adressen befinden und mehr Platz zur Verfügung steht. Wenn man im Memory so scrollt, dass das Byte, auf das der Base-Pointer zeigt, am unteren Rand angezeigt wird, ist der Tooltip unter dem Byte nicht ersichtlich. Damit man trotzdem merkt, dass etwas angezeigt wird, öffnet sich der Text direkt auf dem Byte und wird erst dann nach unten geschoben.

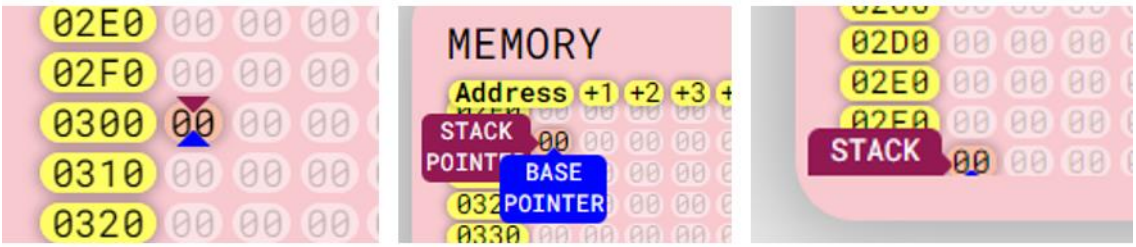


Abbildung 29 Zweite Umsetzung des Stack- und Base-Pointers im Memory mit Tooltips

6.8.2 Instruction-Pointer

Den Instruction-Pointer und die dazugehörigen Bytes haben wir entsprechend der Stack- und Base-Pointer-Bytes dargestellt. Um den Instruction-Pointer von den anderen beiden

Pointern zu unterscheiden, haben wir ihm einen blauen Rand gegeben (siehe Abbildung 30). Die dazugehörigen Instruction-Bytes werden gelb dargestellt, da der Instruction-Pointer in der CPU auch gelb ist.

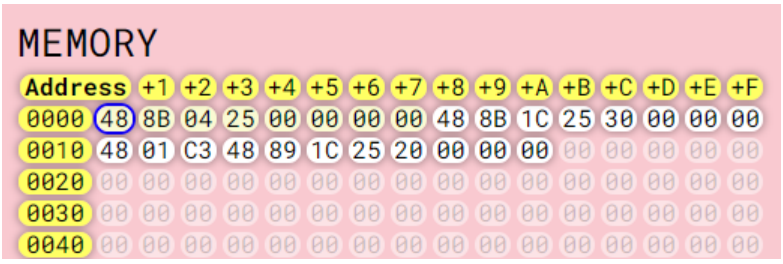


Abbildung 30 Umsetzung des Instruction-Pointers im Memory

Im ersten Usability Test haben die Testpersonen bemerkt, dass es sich um den Instruction-Pointer handelt. Trotzdem haben wir analog zum Stack- und Base-Pointer einen Tooltip hinzugefügt. Wenn alle drei Pointer auf dasselbe Byte zeigen, sind alle drei Pointer Symbole sichtbar (siehe Abbildung 31). Den Tooltip können wir, wie im vorherigen Abschnitt beschrieben, nicht rechts und nicht oben anzeigen. Deshalb lassen wir die Texte so überlappen, dass man alle drei erkennen kann. Beim Instruction-Pointer ist vor allem der Teil “INSTR” relevant, deshalb zeigen wir ihn so hinter dem Base-Pointer, dass dieser Teil lesbar ist. Wie der Base-Pointer öffnet sich auch der Instruction-Pointer Tooltip über dem Byte, und bewegt sich dann nach unten. So kann man den Tooltip auch erkennen, wenn man sich am unteren Rand des Memory befindet.

Abbildung 31 Darstellung aller Pointer und deren Tooltips im Memory

6.9 Change Log

Das Change Log soll zeigen, welche Instruktionen ausgeführt wurden und welche Daten verändert wurden. Während des ersten Usability Tests haben sich die Teilnehmenden einen Wiederholungs-Button gewünscht, mit dem sie einen Schritt erneut ausführen könnten, falls sie verpasst haben, was passiert ist. Als wir ihnen im Anschluss unsere Designidee für den Change Log im Microsoft PowerPoint zeigten, fanden alle, dass dieses Feature ausreiche, um das Geschehene nachzuvollziehen, wenn man etwas verpasst hat.

Danach haben wir begonnen das Change Log umzusetzen. Dazu haben wir die QMenu[16] Komponente von der Quasar GUI Library verwendet. Das Menü wird eingeblendet, sobald man auf den Button klickt und wieder ausgeblendet, wenn man ausserhalb des Menüs klickt. Bei vielen Einträgen wird das Menü automatisch scrollbar. Die Information, was während einer Instruktion geändert wurde, wird in den Services dem **State** Objekt im Attribute **Change-History** hinzugefügt. Bei einer Änderung an diesem Attribut, aktualisiert Vue.js den Inhalt der Change Log Komponente. Solange keine Instruktion ausgeführt wurde, wird ein Text im Change Log angezeigt, dass sich noch nichts geändert habe (siehe Abbildung 32).

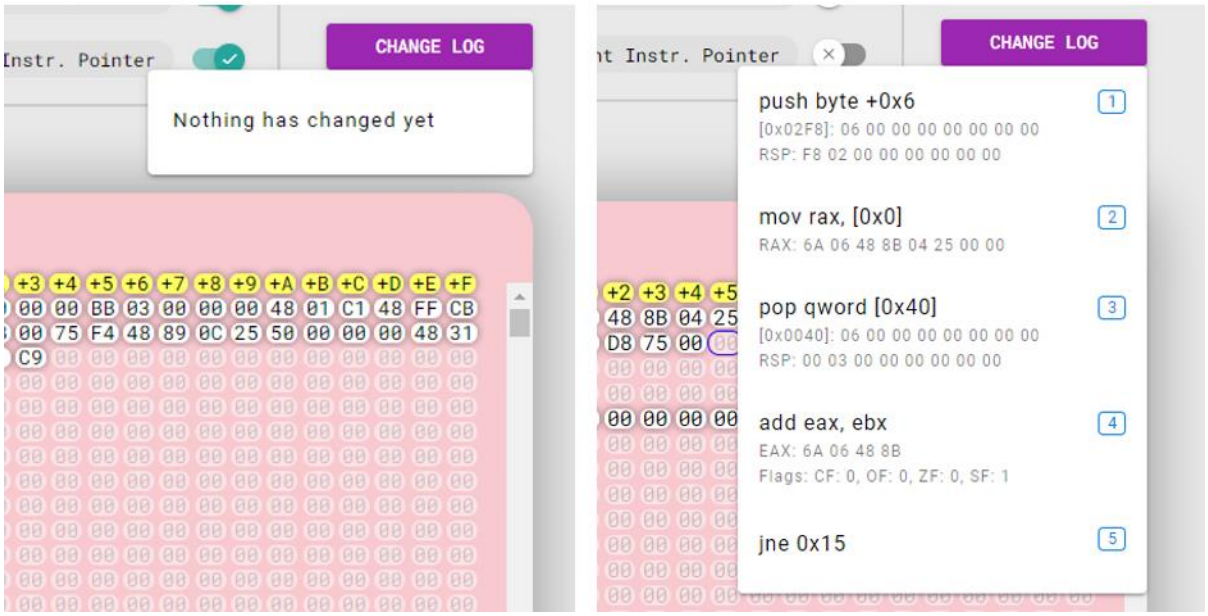


Abbildung 32 Umsetzung des Change Logs

6.10 Assembly Code

Während des ersten Usability Tests ist uns aufgefallen, dass die Teilnehmenden während der Simulation oft den Code, den sie aus einem Dokument mit Beispielprogrammen kopiert haben, geöffnet haben, um nachzusehen, wo sie sich im Programm befinden. Um das zu vereinfachen, haben wir analog zum Change Log ein Menü Button zum Anzeigen des eingegebenen Assembly Code hinzugefügt. In der Studienarbeit haben wir herausgefunden, dass es wichtig ist, dass sich die Assembly Interpretation vom Rest des Simulators unterscheidet, damit die Studierenden nicht das Gefühl haben, der Code werde als Assembly im Memory gespeichert. Deshalb haben wir die Darstellung als Menü, dass man ein- und ausblenden kann für geeignet befunden (siehe Abbildung 33). [1]

[16] Quasar QMenu Komponente: https://quasar.dev/vue-components/menu

Der eingegebene Code wird bei der Navigation vom Code Editor als Base-64-kodierte ASCII-Zeichenkette dem Simulator übergeben. Der Simulator dekodiert den String und gibt ihn dem Assembly Code Menü weiter. Bei der Darstellung des Codes stellte sich die Frage, ob man ihn wie im Editor mit Syntax Highlighting darstellen sollte. Wir haben uns dann aber dagegen entschieden, da man sonst das Gefühl haben könnte, man könne den Code bearbeiten. Ausserdem haben wir einen Tooltip hinzugefügt, der den Hinweis gibt, dass man den Code im Editor ändern kann.

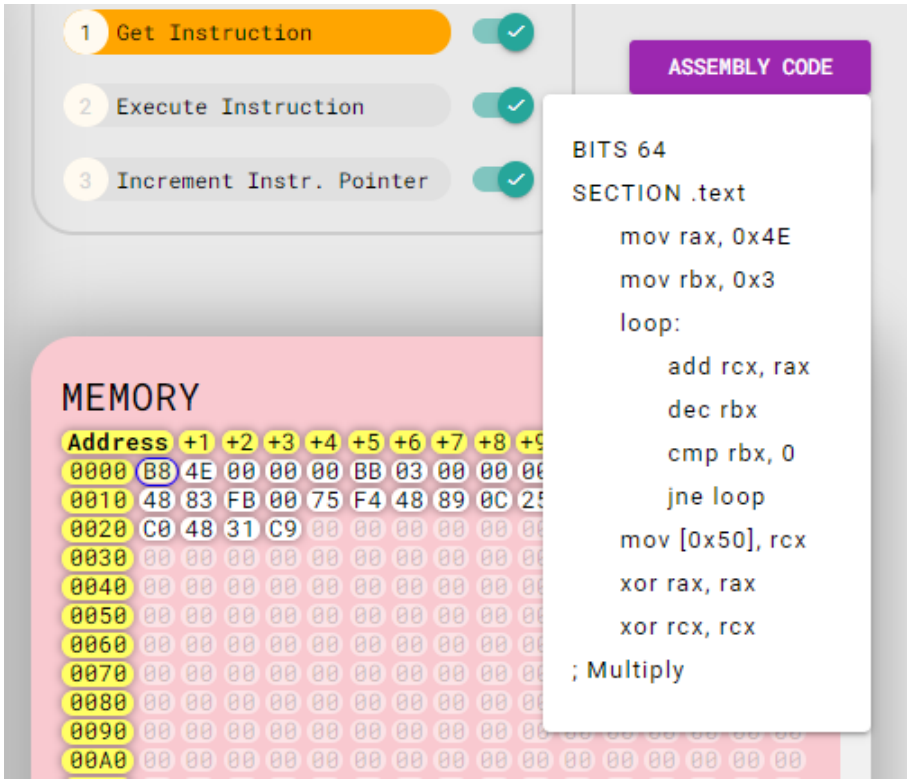


Abbildung 33 Umsetzung der Assembly Code Komponente

6.11 Animationen überspringen

Um die Animationen für einen Schritt ausschalten zu können, haben wir zu jedem Schritt, die in der CPU-Cycle Vue-Komponente dargestellt werden, einen Toggle Button hinzugefügt (siehe Abbildung 34). Da der Platz zu knapp ist, um die Toggle Buttons anzuschreiben, haben wir Tooltips ergänzt, die darauf hinweisen, dass man damit die Animationen pro Schritt ein- oder ausschalten kann. Da der Simulator keinen State über die Seitennavigation hinweg speichert, gehen die Einstellungen verloren, wenn der Benutzende die Seite neu lädt und er muss sie erneut vornehmen.

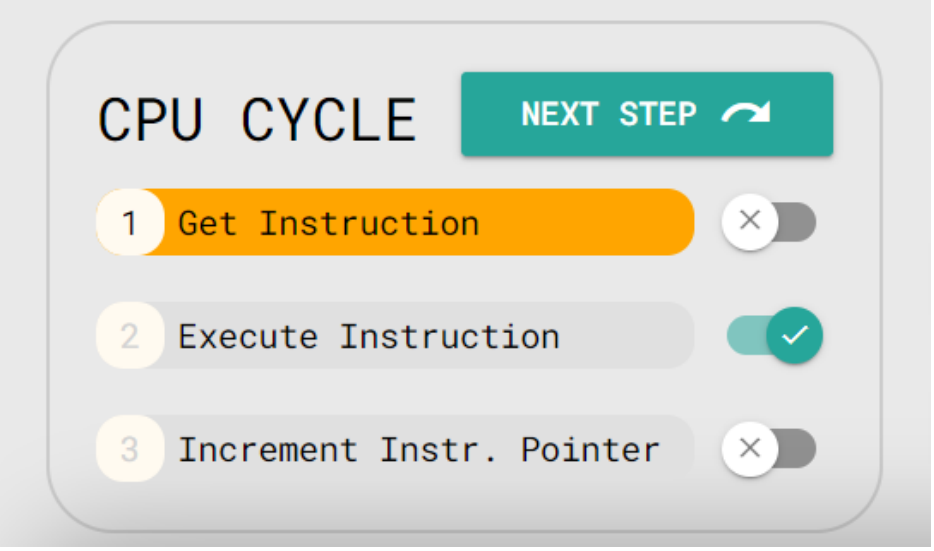


Abbildung 34 Umsetzung der CPU Cycle Box mit Skip Animation Toggle Buttons

6.12 Buttons

In der Studienarbeit hatten wir im Simulator nur einen Button, den “Next Step” Button (siehe Abbildung 35). Wir verwendeten jedoch nur ein Icon ohne Text, so dass dies im damaligen Usability Test von einigen als “einen Schritt überspringen” missverstanden wurde. Wir planten damals dies mit einem Tooltip zu verhindern.

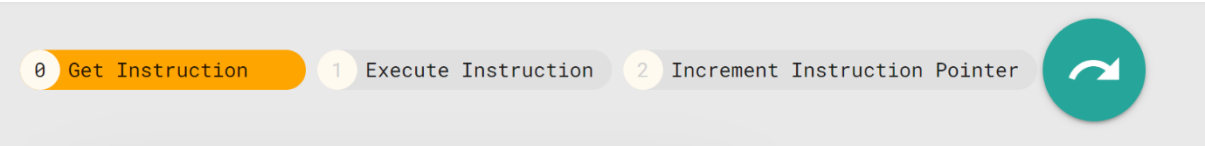


Abbildung 35 Darstellung des "Next Step" Buttons im Prototyp

Während der Bachelorarbeit haben wir weitere Buttons hinzugefügt. Diese haben wir jeweils beschriftet, aber nicht mit einem Icon versehen, da wir keine auf den ersten Blick verständliche Icons gefunden haben. Um die Buttons im selben Design darzustellen, und auch allfällige Unsicherheiten über die Bedeutung des Icons zu beseitigen, entschieden wir uns, den “Next Step” Button zusätzlich zum Icon ebenfalls zu beschriften. Um ihn dennoch von den anderen Buttons abzuheben, haben wir ihn in der CPU Cycle Box platziert (siehe Abbildung 36).

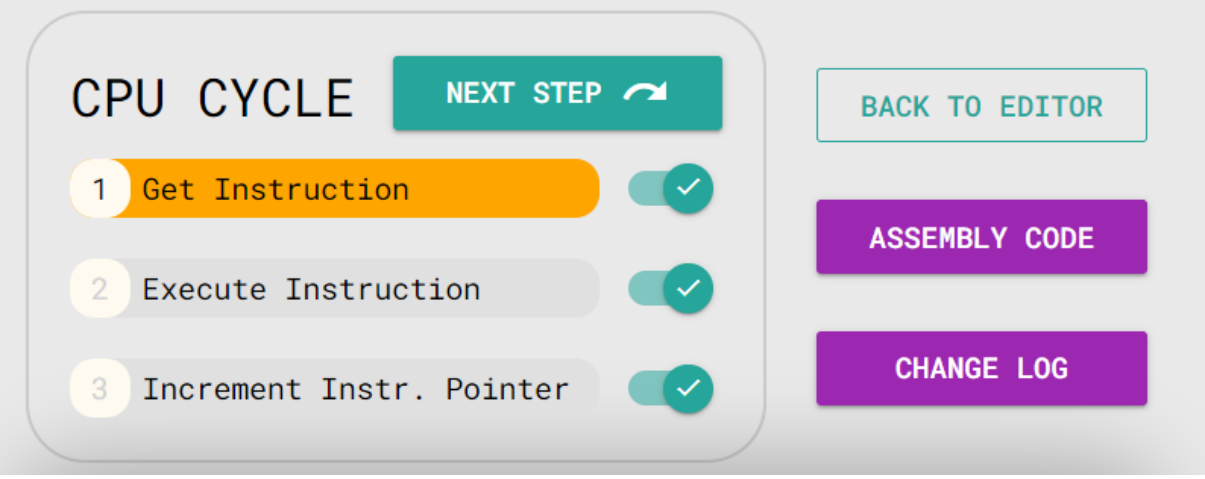


Abbildung 36 Neue Darstellung der Buttons

Die Buttons haben wir mit dem Quasar Framework implementiert. Diese zeigen standardmässig eine Animation, wenn man mit der Maus darüberfährt. Ausserdem reagieren sie, wenn man daraufklickt und pulsieren kurz. Damit im Simulator die Funktionsweise der Buttons klar ist, haben wir bei einigen zusätzlich zur Beschriftung Tooltips hinzugefügt. Dazu haben wir ebenfalls Quasar verwendet.

Eine nicht-funktionale Anforderung war, dass man beim Klicken eines Buttons direkt Feedback erhält. Die Navigation zwischen dem Editor und dem Simulator dauert sehr lange, da im JavaScript die Seite ab- und dann wieder aufgebaut werden muss. Deshalb blenden wir mit Hilfe von Quasar einen Loading Spinner auf den Buttons “Start Program” und “Back To Editor” ein, sobald man daraufklickt (siehe Abbildung 37). Dieser Effekt dauert, bis die Navigation vollendet und die neue Seite geladen wird.

Abbildung 37 Loading Spinner des "Next Step" Buttons

6.13 Einbindung der Lizenzen

In der Studienarbeit war unklar, wie wir unseren Simulator lizenzieren werden, deshalb haben wir ihn damals nicht veröffentlicht. Wir achteten bei der Entwicklung jedoch darauf, nur

Open Source Lizenzen zu verwenden. In der Bachelorarbeit entschieden wir uns für die Lizenz GPLv2 und fügten diese hinzu. Um es anderen leicht zu machen zu erkennen, dass man bei unserer GPLv2 Lizenz kein Upgrade auf die Version 3 machen kann, entschieden wir uns den SPDX Identifier anzugeben. Dieser wurde von der Linux Foundation entwickelt, um ein effizientes und genaues maschinelles Erkennen der Lizenzen zu ermöglichen. [14]

Als wir gegen Ende der Arbeit noch einmal überprüften, ob wir die Lizenzbedingungen der verwendeten Bibliotheken erfüllen, stellten wir fest, dass wir eine Lizenz missachtet hatten. In der Studienarbeit hatten wir uns für passende Fonts und Icons entschieden und diese Base64 kodiert in unserem Projekt eingebunden. Diese wurden jedoch unter Apache 2.0 lizenziert und dürfen nicht in ein GPLv2 Projekt eingebunden werden. [4]

Somit haben wir uns auf die Suche gemacht nach brauchbaren und passend lizenzierten Fonts. Es stellte sich jedoch heraus, dass die Lizenzen der üblichen Fonts nicht kompatibel sind. Eine kompatible Font wäre GNU FreeFont[17] gewesen, die uns jedoch nicht gefallen hat. [4]

Wir entschieden uns deshalb im CSS eine Liste von Fonts anzugeben, von denen der Browser dann die erstbeste, die er lokal installiert hat, auswählt. Dies löste jedoch nur das Problem mit den Fonts, und nicht das der Icons, die wir über Quasar verwendet hatten. Denn diese sind auf keinem Endsystem installiert. Quasar gibt auf der Webseite mehrere Icon Libraries an, die sie unterstützen [15]. Wir stiessen dort auf eine kompatible Bibliothek, die wir dann eingebunden haben. Für das Icon des “Next Step” Button fanden wir dort jedoch keinen passenden Ersatz. Deshalb zeichneten wir dieses in einem Vektor Grafik Editor von Hand leicht verändert nach und fügten dieses ein. Somit haben nun alle Fonts und Icons eine kompatible Lizenz und dürfen so mit dem Produkt verteilt werden.

6.14 Farbkonzept

Farbdesign bedeutet nicht nur, dass man bei der Produktentwicklung schöne Farben auswählt. Ein sehr wichtiger Faktor ist die Accessibility, also dass ein Produkt auch für Menschen bedienbar ist, die ein Problem mit der Farberkennung haben. Beim Entwerfen unseres Farbkonzepts legten wir deshalb grossen Wert darauf, dass die verschiedenen Komponenten von möglichst vielen Studierenden unterschieden werden können.

[17] GNU FreeFont: https://www.gnu.org/software/freefont/sources/index.html

Der Chrome Browser bietet die Möglichkeit die Farben einer Webseite so zu simulieren, wie sie für Menschen mit einer Farbenfehlsichtigkeit aussehen[18]. Wenn man die verschiedenen Farbenfehlsichtigkeitssimulationen unseres GUI in der Abbildung 38 vergleicht, erkennt man, dass die Memory- und CPU-Bereiche, Buttons, Prozessor-Zyklus-Schritte, Pointer und Byteinformationen unterschieden werden können. Sogar ganz ohne Farben kann man alles, bis auf die verschiedenfarbigen Byte Hintergründe unterscheiden. Wir haben bei der Farbwahl mehr Wert auf die Didaktik als auf Ästhetik gelegt, weshalb das Farbkonzept optimierbar ist. Da wir die Farben in einem einzigen Vue-File spezifiziert haben, können sie relativ einfach angepasst werden. Bei einer Überarbeitung in einer Folgearbeit, darf jedoch der didaktische Aspekt für Studierende, die Probleme mit der Farberkennung haben, nicht zu Gunsten der Ästhetik verloren gehen.

[18] Farbenfehlsichtigkeitssimulation im Chrome Browser: https://developer.chrome.com/blog/new-in-devtools-83/#vision-deficiencies

Abbildung 38 Farbenfehlsichtigkeitssimulationen von unserem GUI

6.15 Qualitätssicherung

In diesem Abschnitt wird beschrieben, wie wir eine gute Qualität des Projekts sichergestellt haben. Die erreichten Endergebnisse werden im Kapitel Ergebnisse beschrieben.

6.15.1 Usability

Mit dem ersten Usability Test, den wir in der Mitte des Projekts durchgeführt hatten, konnten wir Fehler erkennen und die Feature Erweiterungen so priorisieren, dass das Endprodukt

benutzbar ist und diejenigen Funktionalitäten anbietet, die für die Verwendung des Simulators in den Übungen benötigt werden. Der genaue Testverlauf ist im Anhang D dokumentiert. Des Weiteren haben wir unsere Ideen und Umsetzungen immer wieder Prof. Stefan Richter und Florian Bruhin vorgestellt und so ebenfalls Feedback zur Benutzbarkeit gesammelt.

6.15.2 Unit- und Systemtests

Wir haben bei den Tests das Vorgehen aus der Studienarbeit übernommen. Das bedeutet, dass keine Unittests für Services geschrieben wurden, die für die Animationen zuständig sind. Es wurden ausserdem im Laufe des Projekts Programme wie zum Beispiel ein Stack Programm hinzugefügt, die beim Ausführen der Tests schrittweise ausgeführt und die resultierenden Änderungen im **State** validiert werden. Zusätzlich wurden wie in der Studienarbeit vor und nach einem Merge Request Systemtests manuell durchgeführt. Dabei werden anhand sorgfältig ausgewählten Instruktionen die Animationen und Funktionalitäten im GUI überprüft. Im Anhang N befindet sich das Protokoll der letzten Durchführung des Systemtests. [1]

6.15.3 Code Analyse

Die Codequalität haben wir auf dieselbe Weise wie in der Studienarbeit sichergestellt. Wir haben wieder Better Code Hub[19] verwendet, um Kriterien, wie kurze und simple Code Units, oder lose gekoppelte Komponenten zu überprüfen. Durch dieses Tool und auch gegenseitige Code Reviews konnten wir Unschönheiten früh erkennen und ein Refactoring vornehmen. Um die Anzahl Codezeilen zu bestimmen haben wir DeepScan[20] verwendet. Dieses Tool ist auf JavaScript spezialisiert und erkennt im Gegensatz zu Better Code Hub auch die Vue-Komponenten. Bei jedem Build hat zudem ESLint[21] eine statische Codeanalyse durchgeführt. [1]

6.15.4 Website Analyse

Um die CPU-Auslastung sowie die Framerate der Applikation zu messen, verwenden wir wie in der Studienarbeit Performance Analysetools, die im Chrome- und Safari-Browser eingebaut sind. Die Ladezeit der Webseite könne wir über den Lighthouse-Benchmark[22] im

[19] Better Code Hub: https://bettercodehub.com/
[20] DeepScan: https://deepscan.io/
[21] ESLint: https://eslint.org/
[22] Lighthouse: https://developers.google.com/web/tools/lighthouse

Chrome-Browser ermitteln. Da wir in dieser Arbeit bei den Performance Problemen aus der Studienarbeit keine Optimierung vorgenommen haben, erwarten wir ähnliche Ergebnisse wie in der Studienarbeit.

7 Ergebnisse

Das Resultat dieser Arbeit besteht aus dem Prozessor-Simulator (siehe Abbildung 39 und Abbildung 40) und einer Anleitung, die im Anhang F zu finden ist. Ausserdem haben wir während der Arbeit Ideen für Erweiterungen im Anhang G dokumentiert und uns erste Gedanken gemacht, wie diese umgesetzt werden können. In diesem Kapitel stellen wir die Ergebnisse der durchgeführten Tests mit dem Endstand des Produkts und unsere Erkenntnisse daraus vor.

Abbildung 39 Screenshot des Endprodukts - Code Editor

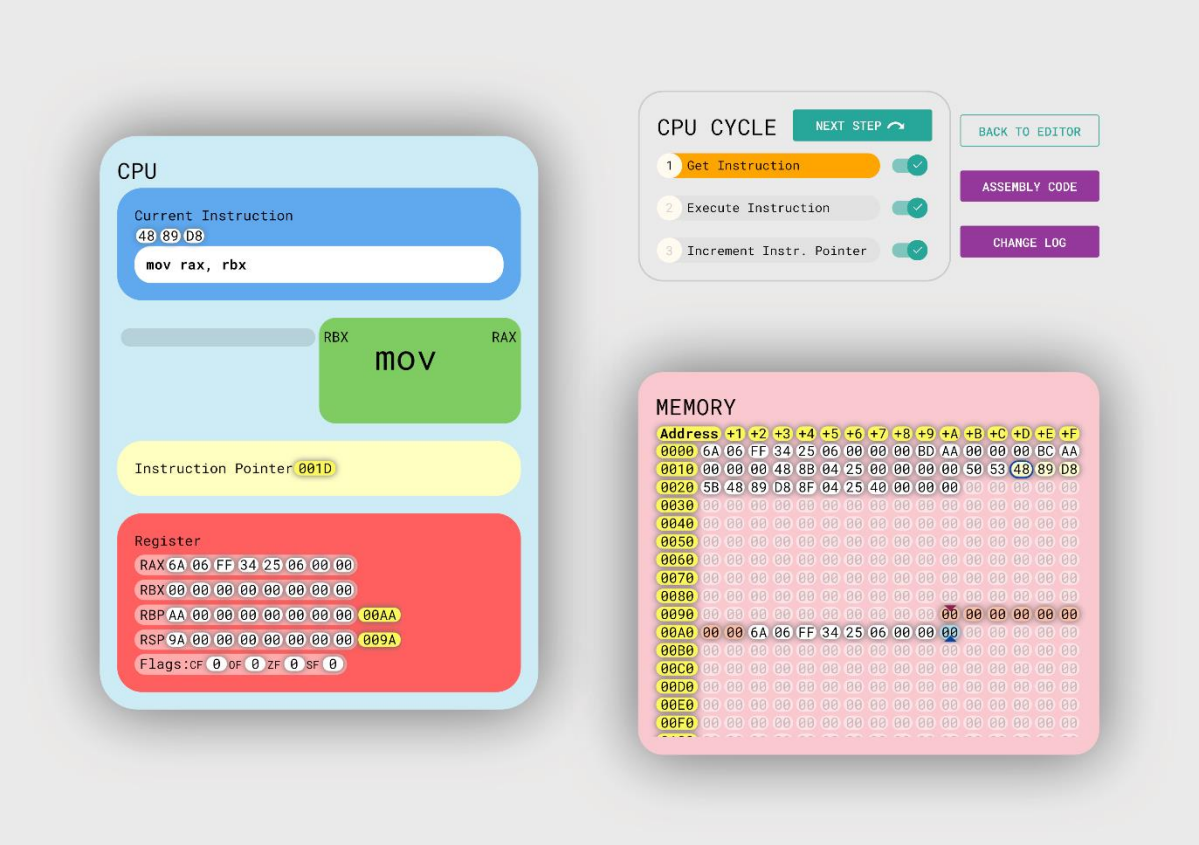


Abbildung 40 Screenshot des Endprodukts - Simulator

7.1 Usability Test

Mit dem Endprodukt führten wir einen zweiten Usability Test durch, an dem fünf Studierende des Moduls Bsys2 teilnahmen. Wir überprüften anhand dieses Tests, ob die Bedienbarkeit gewährleistet ist, ob der Simulator den Studierenden gefällt und ob sie ihn für die Vorlesung geeigneter halten als die Konkurrenzprodukte. Der genaue Ablauf
des Tests ist im Anhang E festgehalten. Ausserdem haben wir dort einige Ideen und Verbesserungsvorschläge der Testpersonen dokumentiert und uns überlegt, wie man diese umsetzen könnte.

7.1.1 Vergleich mit Davis Simulator

Um den Simulator mit den Konkurrenzprodukten zu vergleichen, haben wir uns entschieden, die Studierenden ein Programm in einem fremden Simulator ausführen zu lassen, bevor sie dasselbe in unserem Simulator wiederholten. Es war schwierig einen Simulator für einen fairen Vergleich mit unserem zu finden, da alle, die wir gefunden haben, bei weitem nicht der

Vorlesung gerecht wurden, Zustandsänderungen nicht animiert wurden und die Bedienbarkeit meistens sehr schlecht war. Der Davis Simulator war der Einzige, der eine ähnliche Syntax wie in der Vorlesung anbot und den Prozessor auf einem passenden Abstraktionsniveau darstellt [16].

Trotzdem fiel während des Tests schnell auf, dass unser Simulator für die Studierenden besser verständlich ist. Beispielsweise sind Änderungen im Davis Simulator nicht animiert. Das bedeutet, dass man keine Chance hat zu sehen, dass etwas passiert ist, wenn ein Wert bei einem Zugriff unverändert bleibt. Die Studierenden haben uns als Feedback gegeben, dass die Darstellung im Davis Simulator Sinn ergeben würden, wenn man bereits weiss, was die Instruktion machen soll. Die Bedienbarkeit war im Davis Simulator auch nicht gegeben, da wir oft gefragt wurden, wie man einen einzelnen Schritt vorwärts gehen kann. Während des Tests hat ein Teilnehmer entdeckt, dass man Breakpoints setzen kann. Wir haben davor gar nicht erkannt, dass diese Funktionalität angeboten wird.

Bevor wir unseren Simulator vorgestellt haben, hat keiner der Studierenden bemerkt, dass es sich beim Davis Simulator nicht um unser Endprodukt handelt. Ausserdem hat man beim Öffnen unseres Simulators bei den Studierenden eine gewisse Erleichterung gespürt. Einige haben am Ende des Tests gesagt sie hätten noch viel schlechteres Feedback gegeben, wenn sie gewusst hätten, dass er nicht von uns ist. Es hat sich während des Tests gezeigt, dass sich die Studierenden zwar gewisse Features wie Breakpoints setzen oder Animationstempo ändern wünschen, aber das Verständnis für den Stoff und die Bedienbarkeit bei uns viel höher ist.

7.1.2 Erfüllung der nicht-funktionalen Anforderungen

Die Ziele aus den nicht-funktionalen Anforderungen, dass die Applikation innert fünf Minuten bedienbar ist und ohne Installation in der neusten Version von Chrome auf Ubuntu Linux, MacOS, Windows genutzt werden kann, haben wir erreicht. In den Usability Tests waren alle drei Betriebssysteme abgedeckt und das Starten der Applikation verlief überall problemlos. Im Gegensatz zum ersten Usability Test hat sich die Bedienbarkeit durch den “Back to Editor” Button und dem Assembly Code Menu stark verbessert.

Die nicht-funktionale Anforderung, dass der Prozessor-Zyklus innert dreissig Minuten verstanden wird, haben wir ebenfalls erreicht. Die Studierenden haben die Verständnisfragen, die wir ihnen zu Beginn und während des Tests gestellt haben, im Nachhinein besser und

genauer beantwortet. Wenn sie bei einer Frage unsicher waren, haben sie sofort im Simulator nachgeschaut. Der Stack wurde im zweiten Usability Test besser erkannt und verstanden. Dies liegt einerseits am Stack Beispielprogramm, das zeigt wie der Stack funktioniert und andererseits an den Tooltips und der veränderten Darstellung der Stack- und Base-Pointer Registern.

Bereits im ersten Test waren unsere Animationen für eine Person, die nicht Informatik studiert, verständlich und nachvollziehbar. Die Person hat einen grossen Teil des vermittelten Stoffs verstanden, ohne jemals die dazugehörige Vorlesung besucht zu haben. Das Produkt kann also ohne grosse Einführung von den Studierenden verwendet werden. Somit könnte es zum Beispiel auch zur selbständigen Vorbereitung vor der Vorlesung an die Studierenden ausgehändigt werden.

Wir haben uns Sorgen darüber gemacht, dass unsere Abstraktionen für Studierende, die bereits viele Vorkenntnisse haben unklar oder verwirrend sein könnten. Im Test hat sich jedoch gezeigt, dass diejenigen mit guten Vorkenntnissen sich jeweils überlegt haben, was eine Instruktion macht und sich dann von unseren Animationen bestätigt fühlten. Wir haben also einen guten Mittelweg gefunden, um Personen mit verschiedenen Vorkenntnisse abzuholen.

Die meisten Testpersonen haben sich gewünscht, sie hätten einen solchen Simulator für die Übungen oder zum Lernen auf die Prüfung verwenden können. Um ohne Simulator zu verstehen was der Assembly Code macht, mussten sie Zusatzprogramme verwenden, um die Zustandsänderungen in der Konsole auszugeben. Mit unserem Simulator können diese Informationen visuell nachvollzogen werden. Während des Tests haben die meisten ihre Freude mehrfach zum Ausdruck gebracht und man hat gemerkt, dass sie beim Ausprobieren Spass hatten. Wir denken, dass das alles ist, was eine gute Lernsoftware mit sich bringen sollte.

7.2 Anforderungen

In der Aufgabenstellung wurden Anforderungen gestellt, die wir aus Zeitgründen nicht implementieren konnten. Es handelt sich um Funktionsanforderungen, die wir teilweise bereits in der Studienarbeit als weniger wichtig priorisiert haben und den Studierenden einen kleineren Mehrwert bringen als diejenigen, die wir umgesetzt haben. Die Anforderungen beinhalten das Anzeigen eines Speicherbusses, das Anbieten von festlegbaren Speicherbereichen und

das schrittweise ausblenden von benutzten Bytes. Wir haben bei der Entwicklung jedoch darauf geachtet, dass diese in einer Folgearbeit umsetzbar sind und haben erste Ideen dokumentiert, wie man diese einbauen könnte. Eine Anforderung aus der Aufgabenstellung, die wir zwar umgesetzt haben, die sich jedoch in den Usability Tests als noch nicht ganz ausgereift erwiesen hat, ist das Change Log. Der Inhalt hat nicht den Erwartungen der Studierenden entsprochen. Das Change Log müsste ebenfalls in einer Folgearbeit überarbeitet werden.

Das Hauptziel der Aufgabenstellung, dass das Produkt im Unterricht eingesetzt werden kann, haben wir erreicht. Wir haben eine Möglichkeit erarbeitet herkömmliche Instruktionen aus dem x64-Befehlssatz generisch zu animieren. Somit sind alle Programme aus der Vorlesung im Simulator schrittweise ausführbar. Es werden auch weitere Inhalte aus der Vorlesung abgedeckt, wie zum Beispiel der Stack, und man kann alle Register verwenden. Die Studierenden aus den Usability Tests waren auch der Meinung, dass der Simulator perfekt zur Vorlesung passt und sie hätten sich diesen für ihren Vorlesungsbesuch gewünscht. Das Produkt konnte ausserdem als Open Source Software veröffentlicht werden.

Die Use Cases und die detaillierten Funktionsanforderungen, die wir zu Beginn des Projekts definierten, haben wir fast alle umgesetzt. Einzig beim Use Case “UC03 Ausführung steuern” wurden die Anforderungen “Ausführung am Stück starten und stoppen” und “Animationstempo einstellen” nicht umgesetzt. Die Bedürfnisse der Personas sind im Endprodukt trotzdem abgedeckt. Durch die Funktionalität, Animationen für einzelne Schritte zu überspringen, kommen wir dem Benutzertyp Charlotte entgegen. Und der Typ Andreas kann durch den Step Button und der jetzigen angenehmen Geschwindigkeit den Animationen gut folgen.

7.3 Codeanalyse

In den folgenden Abschnitte stellen wir die Ergebnisse der Codeanalyse Tools vor die im Anhang H genauer ausgeführt sind. Die Analyse mit Deepscan hat uns den bestmöglichen Grade Good verliehen und hat keine Issues gefunden. Sie hat zudem ergeben, dass wir die Lines of Code seit der Studienarbeit verdreifacht haben. Better Code Hub hat unser Projekt jedoch in einer von zehn Kategorien für nicht gut genug befunden. In dieser Kategorie geht es um die lose Kopplung der Komponenten. Die Files **AnimationController**, **MemoryService**, **InstructionService** und **ByteInformationService** verstossen hier gegen die Regeln

des Tools. Laut Better Code Hub liegt dies daran, dass diese Services von anderen Komponenten aufgerufen werden und zudem selbst auch andere Komponenten aufrufen.

Der Vorteil von loser Kopplung der Komponenten ist, dass ein System wahrscheinlich leichter modifiziert und erweitert werden kann. Es gibt verschiedene Design Patterns (zum Beispiel Abstract Factory oder Mediator), die mithilfe von abstrakter Kopplung und Schichtung helfen, lose Kopplung zu erreichen. Wir hätten solche Patterns einsetzen sollen, um zu verhindern, dass man, wenn man zum Beispiel die Unicorn Engine austauschen möchte viele Änderungen vornehmen muss. [17]

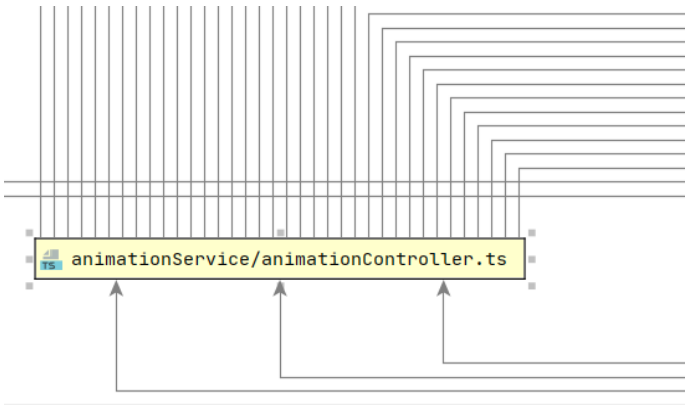


Abbildung 41 Ausschnitt aus dem Klassendiagramm des animationController von WebStorm

Wir haben zwei dieser Klassen mit enger Kopplung genauer untersucht und uns Gedanken gemacht, wie man das Problem beheben könnte. Der **AnimationController** ist die Einstiegsstelle in die **AnimationServices** und wird vom **Controller** aufgerufen. In der Abbildung 41 sieht man einen Ausschnitt aus dem Klassendiagramm, das wir in WebStorm generiert haben, dass der **AnimationController** viele Klassen aufruft. Neben den **AnimationServices** ruft der **AnimationController** auch die **Emulator** Komponente und die **DataServices** auf. Man sollte ein Interface anbieten, über das der **Controller** auf den **AnimationService** zugreifen kann. Somit könnte man den **AnimationService** austauschen, ohne dass man zu viele Änderungen vornehmen muss. Ausserdem könnte man sich überlegen ein Pattern wie Bridge zu verwenden, das aus dem **AnimationController** eine abstrakte Klasse macht und ihn von der Implementierung trennt. Somit könnte man die Implementierung während der Runtime ändern, zum Beispiel wenn man eine andere Animation als die generische, abspielen möchte. [18]

Der **ByteInformationService** liegt in den **DataServices** und wird von drei Komponenten aufgerufen: **AnimationService** (**AnimationController**), **DataService** (**FillDataService**) und

vom **Controller**. Diese rufen verschiedene Funktionalitäten auf. Der **AnimationController** möchte die Pointer updaten und der **Controller** möchte wissen, ob das Byte, auf das der Instruction Pointer zeigt, zu den "**used**"-Bytes gehört. Das Single-Responsibility-Prinzip wird hier verletzt und man sollte sich hier überlegen, ob man die Funktionalitäten in verschiedene Klassen aufteilen könnte [19]. Nach dem Aufteilen kann man noch einmal überlegen, welche Funktionalität in welcher Komponente positioniert werden soll. Somit könnte die starke Kopplung reduziert werden.

7.4 Architektur

In den folgenden Abschnitten wird die Erfüllung der nicht-funktionalen Anforderungen bezüglich der Architektur und der Performance geschildert.

7.4.1 Modularer Aufbau der Applikationskomponenten

Obwohl wir uns bei der Implementierung über die Architektur Gedanken gemacht haben und diese immer wieder überarbeitet haben, sehen wir Verbesserungspotential. Wie bereits im Resultat des Tests bei Better Code Hub erwähnt, sind die Komponenten zu stark gekoppelt. Ein weiteres Beispiel ist der **emulatorService** auf den fast alle Klassen aus den **dataServices** zugreifen. Der Grund hierfür ist, dass der Enum, der die **RegisterID** beinhaltet im **emulatorService** liegt. Diese würde man besser in die **Interfaces** oder **Helper** verschieben, da alle Schichten in der Architektur darauf zugreifen. Ausserdem könnte man im **fillDataService** alle Daten, die man benötigt aus dem **Emulator** auslesen und die resultierenden UInt8Arrays an die anderen **dataServices** weitergeben, so können sich diese auf das Abfüllen der Daten in die Datenstruktur des **States** konzentrieren und müssten nicht alle auch auf den **Emulator** zugreifen.

Das **Program** Objekt diente in der Studienarbeit lediglich dazu, die Parameter bei Aufrufen des **Controllers** zu reduzieren. Da es nun die Konfiguration (Memory Grösse, Startadresse des Codes etc.) enthält, wird es vom **Controller** weitergereicht. Dies könnte man jedoch eleganter lösen, indem man die Konfiguration als globale Variablen speichert und so von den Services abfragen kann. So könnte man neue Daten zur Konfiguration hinzufügen, ohne das **Program** Objekt verändern zu müssen.

Wir haben die nicht-funktionale Anforderung nicht erreicht, dass die Applikationskomponenten modular aufgebaut werden sollen. Das Hinzufügen neuer Funktionalitäten gegen Ende

des Projekts war jedoch sehr einfach und schnell möglich. Dies verdanken wir den Refactorings, die wir während des Projekts immer wieder vorgenommen haben. Wir konnten beispielsweise einige Fälle korrigieren, in denen die Funktionen in den Files komplett verschiedene Aufgaben hatten und das Single-Responsibility-Prinzip [19] verletzten. Wir haben beispielsweise auch darauf geachtet, möglichst treffende Namen für die Funktionen zu wählen. Ausserdem haben wir die Konfiguration des Simulators, die Beispielprogramme und die CSS-Farben so gestaltet, dass man sie möglichst einfach anpassen kann.

7.4.2 Website-Analyse

Um zu testen, ob die Architektur von unserem Projekt die Anforderungen erfüllt, testen wir verschiedene Performance-Aspekte. Im folgenden Abschnitt werden die Resultate inklusive unseren Interpretationen dazu erläutert. Die genauen Ergebnisse sind im Anhang I zu finden.

Die Grösse des HTML-File beträgt 7 MB was sich innerhalb der von uns gesetzten Limite von 15 MB befindet. Die Seitenladezeit wurde mit dem Lighthouse-Benchmark vom Chrome-Browser simuliert und beläuft sich auf 6 Sekunden, wobei die Seite innerhalb von 6.5 Sekunden interaktiv ist. Dieses Resultat ist schneller als die 10 Sekunden, die wir in den nicht-funktionalen Anforderungen festgelegt haben. Die Applikation kann also ohne Probleme, in einer Folgearbeit, um weitere Libraries erweitert werden, ohne dass der Benutzende zu lange auf den Start der Applikation warten muss.

Die Auslastung der Hardware liegt auch im Rahmen unserer nicht-funktionalen Anforderungen. Die CPU-Nutzung vom Chrome-Prozess beträgt maximal 18% der Systemauslastung, wobei Safari eine durchschnittliche Auslastung von 19% zeigt. Beides liegt unterhalb von der festgelegten Grenze von 20% der maximalen Systemauslastung.

Die Framerate der Applikation wurde mit Hilfe von mehreren Tools in Chrome und Safari untersucht. Sobald eine Animation im Gang ist, ist die Anforderung an die Framerate erfüllt und liegt über 35 Frames pro Sekunde. Sie bricht jedoch vor der Animation zusammen, was für uns jedoch nicht sichtbar ist, da die Animationen flüssig erscheinen. Somit sind die Anforderungen für uns erfüllt.

Ein weiterer wichtiger Punkt ist, dass alle Benutzereingaben direkt ein Feedback geben. Alle Interaktionselemente wurden mit Hilfe der Quasar GUI Library nach diesem Prinzip gebaut,

und so muss sich der Benutzende nie fragen, ob sein Klick akzeptiert wurde oder nicht, da er sofort ein Feedback erhält.

Die DOM-Grösse blieb seit der Studienarbeit unverändert, was die Animations- und Darstellungsleistung beeinträchtigen könnte. Den Verbesserungsvorschlag, den wir in der Arbeit geschildert haben, könnte man immer noch so umsetzen. Da die Animationen im Prozessor-Simulator flüssig erscheinen, ordneten wir diesen Punkt mit einer Tiefen Priorität ein. [1]

8 Fazit und Ausblick

In dieser Arbeit haben wir gezeigt, wie wir einen Prozessor-Simulator entwickelt haben, der dem State-of-the-Art entspricht. Das resultierende Produkt baut auf bereits bewährten Libraries auf und erweitert diese um unser innovatives Design zu einer verständlichen Lernsoftware. Wir haben mit unserem Produkt das wichtigste Ziel erreicht: der Simulator kann im Unterricht eingesetzt werden und trägt zum besseren Verständnis über die Arbeitsweise des Prozessors bei.

Durch den Einbau der Nasm Syntax können die Studierenden ihren Code sowohl im Simulator als auch direkt auf dem Betriebssystem ausführen. Dieser Faktor und auch unsere Abstraktion, die verständlich und zugleich nahe an der Realität liegt, tragen stark zum Lernprozess bei. Im Gegensatz zu den bestehenden Simulatoren werden bei unserem die Zustandsänderungen mit Hilfe von Animationen ansprechend dargestellt und es ist klar, was die Animation ausgelöst hat.

Die Studierenden, die unseren Simulator getestet haben, sind begeistert von seinem Aussehen und seiner Funktionsweise. Sie hätten sich so einen Simulator für ihren Vorlesungsbesuch gewünscht und erkannten den Vorteil, dass sie Zustandsänderungen nun grafisch sehen können und diese nicht mehr in der Konsole ausgeben müssen. Die Bedienbarkeit ist selbsterklärend und die Studierenden können den Simulator innert Sekunden starten. Er könnte sogar für eine erste selbständige Einführung in die Vorlesung eingesetzt werden.

Leider können wir aus Zeitgründen nicht alle Anforderungen aus der Aufgabenstellung und den nicht-funktionalen Anforderungen erfüllen. Zum Beispiel haben wir noch kein Error Handling implementiert und die Modularität der Architekturkomponenten könnte verbessert werden. In der Aufgabenstellung werden noch weitere Funktionalitäten wie die Darstellung des Speicherbusses und das byteweise Lesen der Instruktion durch die CPU gewünscht. Der Simulator bietet zudem viele Erweiterungsmöglichkeiten. Zum Beispiel könnte man eine Komponente einfügen, die den Maschinencode im Memory nach Assembly übersetzt und das Setzen von Breakpoints ermöglicht. Somit könnte man die Instruktionen bis zum nächsten Breakpoint am Stück ausführen. Ausserdem könnte man für Instruktionen, wie Jump oder Call eine spezifische Animation anbieten, damit diese noch verständlicher werden. Wir haben bei der Entwicklung darauf geachtet, dass der Simulator in einer Folgearbeit erweiterbar ist und Ideen zur Umsetzung im Anhang G und Anhang E festgehalten.

Da wir bei der Weiterentwicklung des Prototyps aus der Studienarbeit darauf geachtet haben, dass die umgesetzten Funktionalitäten einen möglichst grossen Mehrwert für die Studierenden bieten, hat der Prozessor-Simulator bereits im jetzigen Zustand die Produktreife erreicht. Er kann nun in der Vorlesung eingesetzt werden und bei Studierenden mit unterschiedlichen Vorkenntnissen das tiefen Verständnis des Inhalts fördern.

9 Verzeichnisse

9.1 Literaturverzeichnis

[1] E. I. Schmidli und Y. Boillat, «Grafischer Prozessor-Simulator,» OST Ostschweizer Fachhochschule, Rapperswil, 2020.

[2] GNU, «GNU General Public License, version 2,» 04. Oktober 2017. [Online]. Available: https://www.gnu.org/licenses/old-licenses/gpl-2.0. [Zugriff am 15. Juni 2021].

[3] Free Software Foundation, «Apache License, Version 2.0,» Januar 2004. [Online]. Available: Available: https://directory.fsf.org/wiki/License:Apache-2.0. [Zugriff am 15. Juni 2021].

[4] GNU, «Various Licenses and Comments about Them,» 07. Mai 2021. [Online]. Available: https://www.gnu.org/licenses/license-list. [Zugriff am 15. Juni 2021].

[5] S. Richter, Vorlesung Betriebssysteme 1: Funktionen, Rapperswil: OST Ostschweizer Fachhochschule, 2020.

[6] N. A. Quynh, «Capstone Disassembler Engine test_x86.c,» 05. Februar 2019. [Online]. Available: https://github.com/aquynh/capstone/blob/4457d451aad63ed7ac4ef259200d165d157f1554/tests/test_x86.c. [Zugriff am 2021 Juni 15.].

[7] W3C CSS Working Group, «CSSOM View Module,» 09 Juni 2021. [Online]. Available: https://drafts.csswg.org/cssom-view/#dom-element-scrollintoview. [Zugriff am 15 Juni 2021].

[8] Emscripten Contributors, «About Emscripten,» 2015. [Online]. Available: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html. [Zugriff am 15. Juni 2021].

[9] A. Viau, «TweetX86,» 13. September 2018. [Online]. Available: http://twt86.co. [Zugriff am 15. Juni 2021].

[10] A. Viau, «TweetX86 readme.md,» 13. September 2018. [Online]. Available: https://github.com/AntoineViau/tweetx86#build . [Zugriff am 15. Juni 2021].

[11] Emscripten Contributors, «File System API,» 03. Juni 2021. [Online]. Available: https://emscripten.org/docs/api_reference/Filesystem-API.html#file-systems. [Zugriff am 15. Juni 2021].

[12] Intel Corporation, «Opcode Extension Tables,» in *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, 2021, pp. A-18 Vol. 2D.

[13] K. Lejska, «X86 Opcode and Instruction Reference,» 18. Februar 2017. [Online]. Available: http://ref.x86asm.net/coder64.html. [Zugriff am 4. Juni 2021].

[14] The Linux Foundation, «SPDX License List,» 20 Mai 2021. [Online]. Available: https://spdx.org/licenses/. [Zugriff am 06. Juni 2021].

[15] R. Stoenescu, «Icon,» Quasar, 19 Mai 2021. [Online]. Available: https://quasar.dev/vue-components/icon . [Zugriff am 15. Juni 2021].

[16] J. Beránek, «Davis: x86 assembly debugger,» Juli 2017. [Online]. Available: https://kobzol.github.io/davis/. [Zugriff am 01 06 2021].

[17] E. Gamma, R. Helm, R. Johnson und J. Vlissides, «1.6 How Design Patterns solve design problems,» in *Design Patterns: Elements of Reusable Object-Oriented Software. [2nd print.]*, Reading Mass, Addison-Wesley, 1995, p. 24 ff..

[18] E. Gamma, R. Helm, R. Johnson und J. Vlissides, «Bridge,» in *Design Patterns: Elements of Reusable Object-Oriented Software. [2nd print.]*, Reading Mass, Addison-Wesley, 1995, pp. 151 - 154.

[19] R. C. Martin et al., «The Single Responsibility Principle [17th print.],» in *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River NJ, Prentice Hall, 2009, p. 138 ff..

9.2 Abbildungsverzeichnis

9.3 Tabellenverzeichnis

Anhang A Anforderungen

Anforderungen

Persona

Andreas (21)[23]:
Andreas hat nach seiner Informatiklehre begonnen, an der OST Informatik zu studieren. Zusammen mit seinen Freunden besucht er das Grundlagenmodul Bsys1. Er möchte dort erfahren, wie der Prozessor seine Programme ausführt. In der Vorlesung der zweiten Woche ist es so weit und er wird in das Prozessormodell eingeführt. Er hat jedoch Mühe der Vorlesung zu folgen, da es ihm schwerfällt, das neu Gelernte richtig einzuordnen.

Ziel im Prozessor-Simulator:
Andreas ist froh, wenn er nicht mit Informationen überladen wird, damit er schnell einen Überblick über die Komponenten erhält. Er möchte Ausführungen der Code Beispiele aus der Bsys1 Vorlesung sehen, um den vermittelten Stoff besser zu verstehen. Ausserdem möchte er in seinem Tempo vorwärts gehen, da er noch etwas Mühe hat, die einzelnen Schritte nachzuvollziehen.

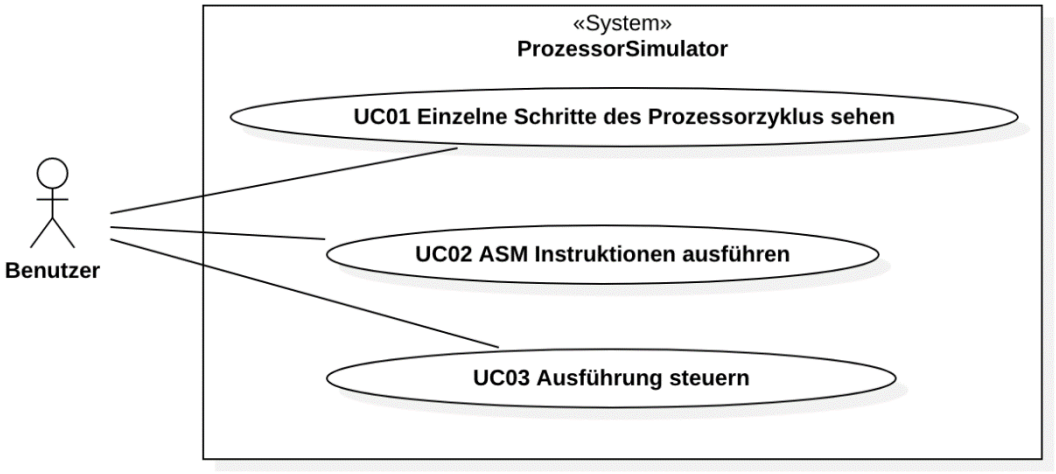
Charlotte (22)[24]**:**
Nach ihrer Lehre als Elektronikerin hat sich Charlotte für ein Informatik Studium an der OST entschieden. Ihr Lieblingsmodul ist Bsys1, da sie dort schon viele Vorkenntnisse hat. Dementsprechend ist sie ihren Mitstudierenden weit voraus. Ihr Ziel ist es, ihre Kenntnisse in Hardwarenaher Programmierung zu vertiefen. Deshalb tüftelt sie gerne in den Übungen an ihrem Assembly Code.

[23] Bild von Andreas wurde erstellt auf https://thispersondoesnotexist.com/
[24] Bild von Charlotte wurde erstellt auf https://thispersondoesnotexist.com/

Ziel im Prozessor-Simulator:

Charlotte lässt im Prozessor-Simulator selbst geschriebene Programme aus der Bsys1 Übung laufen, um ihren Lösungsansatz zu validieren. Um sich nicht zu langweilen, möchte sie gewisse Animationen, die sie bereits verstanden hat, überspringen können oder schneller ausführen.

Use Cases



UC01 Einzelne Schritte des Prozessor-Zyklus sehen

Als Benutzer möchte ich die einzelnen Schritte des Prozessor-Zyklus sehen, um besser zu verstehen, was im Prozessor genau passiert.

Id	**Titel**
01	Aktuelle Instruktion in Nasm Syntax sehen
02	Geänderte Daten durch Animation hervorheben
03	Instruction Pointer sehen
04	Arbeit der CPU sehen
05	Zustandsübergänge graphisch ansprechend sehen
06	Alle Register zur Verfügung haben
07	Bessere Übersicht haben durch ausgeblendete/ausgegraute Elemente
08	Änderungsverlauf der Register sehen können

UC02 ASM Instruktionen ausführen

Als Benutzer möchte ich mein eigenes Programm eingeben und ausführen, um die verschiedenen Instruktionen besser zu verstehen.

Id	**Titel**
09	Demo Instruktion starten
10	ASM Instruktionen eingeben

UC03 Ausführung steuern

Als Benutzer möchte ich die Ausführung steuern können, um in meinem Tempo die Animationen zu sehen.

Id	**Titel**
11	In Schritten vorwärts gehen
12	Ausführung am Stück starten und stoppen
13	Animationstempo einstellen
14	Wiederholende Schritte überspringen (z.B. Get Instruction)

Nicht-funktionale Anforderungen

Maintainability

Nr	Anforderung	Beschreibung
01	Testability	Das Testing kann auf Entwickler- und Benutzerebene durchgeführt werden.
02	Testability	Alle Test-Cases müssen erfolgreich sein und reviewed werden, bevor die Code Änderungen akzeptiert werden.
03	Modifiability	Die Applikation kann angepasst werden, ohne das produktive System zu beeinflussen
04	Modularity	Die Applikationskomponenten werden modular aufgebaut, um einen schnellen und einfachen Austausch zu ermöglichen, damit neue Technologien rasch eingebaut werden können.

Security

Nr	Anforderung	Beschreibung
05	Integrity	Wenn die Applikation von der originalen Plattform heruntergeladen wird, kann man sicher sein, dass sie nicht von Dritten modifiziert wurde.
06	Confidentiality	Es werden keine persönlichen Daten erfasst oder versendet.

Performance

Nr	Anforderung	Beschreibung
07	Resource Behaviour	Die Applikation darf in ihrem Chrome Prozess auf dem Zielsystem (Laptop mit CPU: Intel i7 8565U, 16GB RAM, Bildschirmauflösung: 1920 x 1080, Windows 10) eine durchschnittliche Gesamt-CPU-Nutzung von 20% nicht überschreiten.
08	Resource Behaviour	Die Applikation hat eine Downloadgrösse von maximal 15 MB
09	Time Behaviour	Die Applikation startet innerhalb von 10 Sekunden nach dem Download und nimmt Benutzereingaben entgegen.
10	Time Behaviour	Nach einer Benutzereingabe z.B. Drücken des Start-knopfes gibt die Applikation sofort ein Feedback.
11	Time Behaviour	Die Animationen sind flüssig und stocken nicht. Die Animationen laufen mindestens mit 30 Frames pro Se-kunde.

Usability

Nr	Anforderung	Beschreibung
12	Availability	Die Applikation ist offline lauffähig
13	Learnability	Informatik Studierende sollen die Applikation nach maximal 5 Minuten verstehen und verwenden können.
14	Learnability	4 von 5 Informatik Studierende sollen nach dem Benutzen der Applikation für 30 Minuten verstehen, wie ein Prozessor-Zyklus abläuft
15	Accessibility	Das Produkt kann auch von Menschen, die ein Problem mit der Farberkennung haben, bedient werden.

Portability

Nr	Anforderung	Beschreibung
16	Installability	Das Produkt kann ohne Installation in der neusten Version von Chrome auf Ubuntu Linux, MacOS, Windows genutzt werden.

Anhang B Verwendete Instruktionen in Bsys1

Instruktionen Bsys1

(ohne `.inc.asm` und `.test.asm`)

```
adc
add
and
call
clc
cmp
dec
dq
inc
jmp
jne
jnz
jz
leave
mov
or
pop
push
resd
ret
sbb
sub
xor
```

Anhang C Architektur Dokumentation

Architektur Dokumentation

In diesem Dokument werden die Architektur Komponenten und wichtige Objekte genauer beschrieben. Ausserdem werden die Funktionsweise und Umsetzung gewisser Funktionalitäten genauer ausgeführt. Dieses Dokument dient als Ergänzung zu den Informationen, die wir in den Kapiteln Lösungskonzept und Umsetzung im Bericht erläutern.

Architektur

Im folgenden Komponentendiagramm werden die Beziehungen zwischen den Komponenten dargestellt. Die Komponenten Interfaces und Helper wurden für eine bessere Übersicht weggelassen. Diese werden von fast allen Komponenten aufgerufen. Bei den gross geschriebenen Elementen handelt es sich um eine Klasse, die kleingeschriebenen beinhalten mehrere Klassen. Türkis steht für die GUI, weiss für die Service und violett für die Library Schicht. In der zweiten Abbildung erkennt man die Architekturschichten besser.

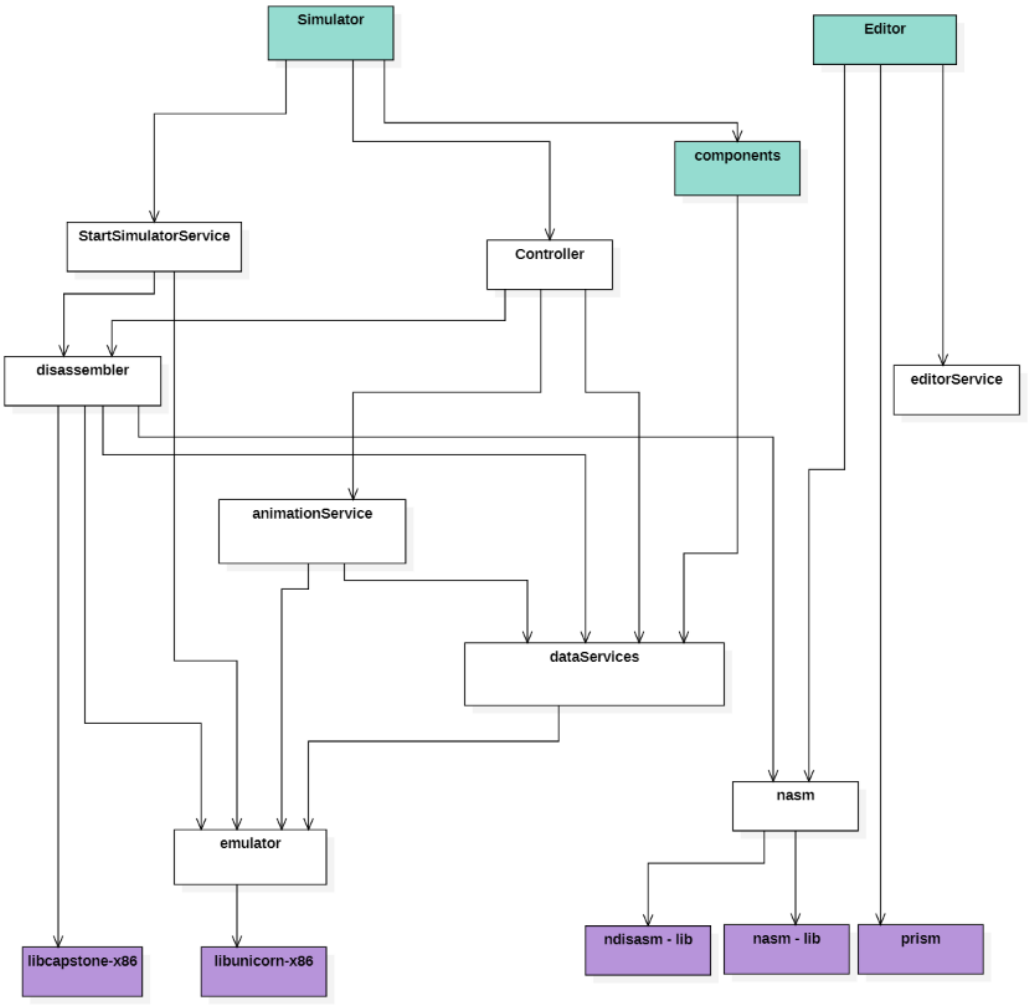


Abbildung C 1 Komponentendiagramm

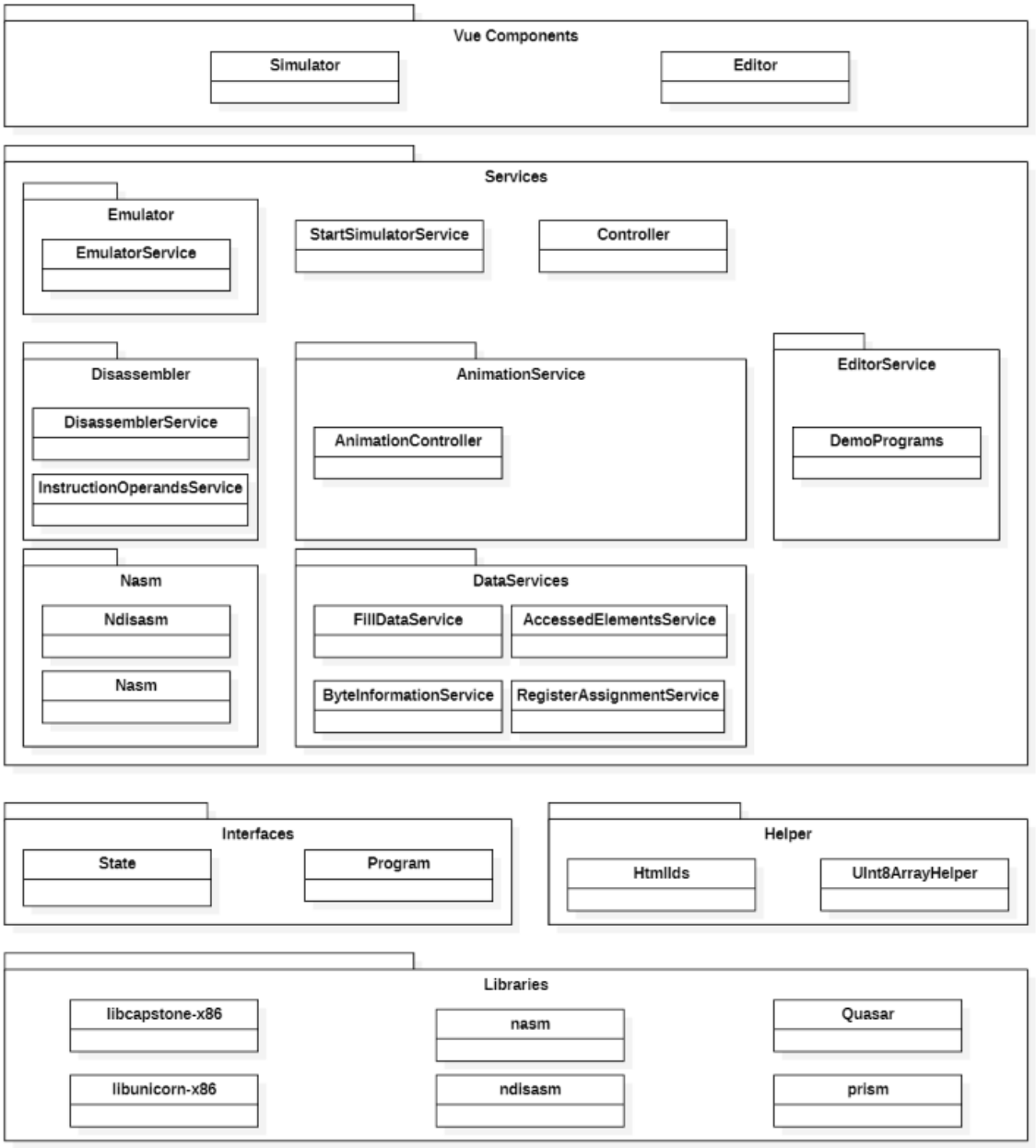


Abbildung C 2 Übersicht über die Architekturschichten mit den wichtigsten Klassen

Aufbau GUI

Die Simulator und die Editor-Vue-Komponente werden vom Router dynamisch zum HTML in App.vue hinzugefügt, je nachdem, ob man sich auf dem Pfad “/editor” oder “/simulator” befindet. In App.vue werden auch die Komponenten aus “colorAndLayout”, die CSS-Variablen enthalten. Zum Beispiel werden in der Komponente Colors allen Farben, die verwendet werden, Variablen Namen zugewiesen, um ein schnelles Anpassen des Farbkonzepts zu ermöglichen. In der Komponente Layout werden die Intensitäten der Schatten und weitere

CSS Stylings vorgenommen, die für den gesamten Simulator gelten. In der Scaling Komponenten, wird je nach Fenstergrösse das Padding, BorderRadius und die Grösse der Elemente angepasst. So wird der Simulator auch noch auf kleinen Screens ansprechend und lesbar dargestellt.

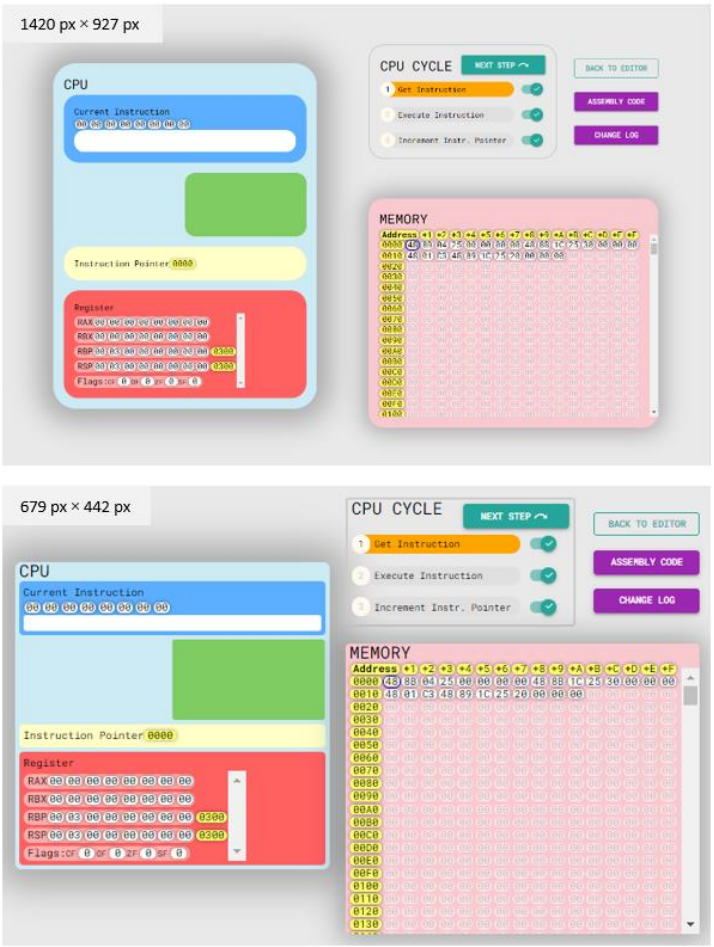


Abbildung C 3 Verschiedene Darstellungen des GUI je nach Bildschirmgrösse

Unser jetziges Responsive Design ermöglicht es, den Simulator auch auf kleinen Bildschirmen anzuzeigen. Es ist jedoch noch nicht möglich diesen auf Mobile Phones darzustellen. Grund dafür ist, dass die Schriftart auf dem kleinen Display unleserlich wird. Ausserdem müsste man die Controls grösser anzeigen, dass man sie mit dem Finger drücken kann. Es würde jedoch nichts im Wege stehen, das Design zu überarbeiten, dass man den Simulator auch auf dem Handy leserlich abspielen könnte.

Editor

Der Editor ist eine Vue-Komponente. Diese Beinhaltet einen Editor, der von prism übernommen wurde. Ausserdem beinhaltet der Editor eine Vue Komponente LicenseButton der alle im Projekt verwendeten Lizenzen anzeigt. Klickt man auf den “Start Program” Button wird der Assembly Code vom Nasm übersetzt. [...]

In der Komponente LicenseButton wird ein Menü dargestellt, dass beim Öffnen Lizenzen anzeigt. Diese stammen vom Array aus licensesToShow. Pro Eintrag kann man einen Namen angeben und den Text der Lizenz.

Demo Programme anpassen

Die Beispielprogramme, die man im Editor mithilfe von Buttons wählen kann, stammen vom File demoPrograms aus dem editorService. Um die Programme anzupassen, kann man die Objekte aus dem demoPrograms Array anpassen. Das Attribut Label wird auf dem Button im GUI angezeigt. Im Array kann man auch ein weiteres Objekt hinzufügen, um einen weiteren Button anzuzeigen.

Simulator GUI

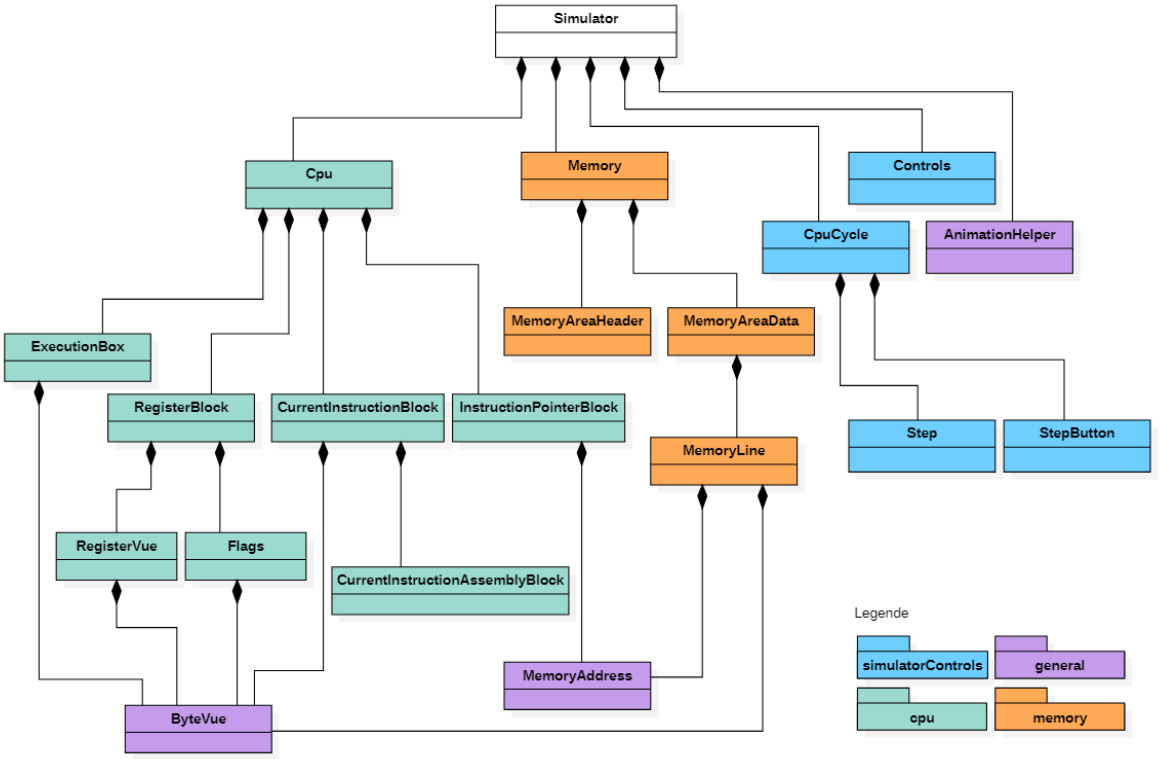


Abbildung C 4 Verschachtelung der Simulator Vue-Komponenten

Die Vue Komponenten im Simulator sind hierarchisch verschachtelt. Sie sind aufgeteilt in “simulatorControls”, “cpu”, “memory” und “general”. Die Simulator Vue-Komponente kommuniziert mit dem “Controller” und dem “StartSimulatorService” aus den Services. Sie erhält so das State Objekt, dass sie an ihre Vue-Komponenten weitergibt, die dann die Zustände darstellen. Wenn “Next Step” geklickt wird, teilt das die Simulator-Komponente dem Controller mit. Sobald die Services das State Objekt aktualisieren, passt Vue in den Komponenten die dargestellten Daten entsprechend an. Wenn die “nextStep” Funktion false zurückgibt, indiziert dies, dass es sich um den letzten Schritt gehandelt hat. Dann wird der “Next Step” Button mit einem “Reload” Button ersetzt. Dieser gibt, wenn er geklickt wird, denselben Event weiter, wie der “Next Step” Button, da der Simulator jedoch weiss, dass das Programm beendet ist, führt er einen Reload der Seite aus.

General Komponenten

In “general” befinden sich die Komponenten ByteVue und MemoryAddress, die von verschiedenen Komponenten verwendet werden. Ausserdem befindet sich dort die Komponente AnimationHelper. In dieser werden die Attention und Move Animationen ausgeführt,

sobald in den AnimationServices die HTML Klassen gesetzt werden. Zum Beispiel wird, wenn die Klasse “moveByte” auf einem Element gesetzt wird, dieses an die zuvor gesetzten Koordinaten verschoben.

In der Komponente ByteVue werden die Bytes dargestellt. Je nachdem welche Informationen die Services im Objekt ByteInformation angeben, werden die Byte anders dargestellt. Zum Beispiel werden Bytes, die nicht zu den “usedBytes” gehören, leicht ausgeblendet. In der ByteVue Komponente werden ausserdem der Stack- und Base-Pointer speziell dargestellt. Ausserdem werden diese animiert, sobald die CSS-Klasse “basePointer” oder “stackPointer” neu gesetzt wird, damit man die Änderung besser sehen kann.

CPU Komponenten

Die CPU Komponenten entsprechen dem Block, der im Simulator links dargestellt wird. Das wären der CurrentInstructionBlock mit der Assembly und Byte Darstellung, der Instruction Pointer und der RegisterBlock, der die Register und Flags darstellt. Des Weiteren wird die Execution Box dargestellt, die der CPU-Box in unserem Konzept entspricht. Diese bewegt sich nach links, sobald die AnimationServices die CSS-Klasse “magicBoxLeft” setzen.

Memory Komponenten

Die Memory Komponenten stellen den rechten Block im Simulator dar. Sie setzen sich zusammen aus einem Header, der die Byte Offsets darstellt, und den Daten. Die Daten werden in Memory Lines dargestellt, die wiederum eine Adresse und Bytes aus den General Komponenten besitzen.

SimulatorControls Komponenten

Hier befinden sich die CPU-Cycle Komponente und die Buttons. In der Komponente Controls befinden sich die drei Buttons “Back to Editor”, “Assembly Code” und “Change Log”. Der “Back to Editor” Button übergibt dem Vue-Router den Assembly Code und navigiert zurück zum Editor, wo der Code dann eingelesen und dargestellt wird. Die anderen beiden Buttons öffnen jeweils ein Menü.

Simulator Services

StartSimulatorService

Der StartSimulatorService wird von der Simulator-Vue-Komponente aufgerufen, um den Emulator und den Disassembler zu initialisieren. Als Resultat gibt der Service das Program Objekt zurück, dass dann dem Controller mitgegeben wird. In diesem Service kann man die Konfiguration von Unicorn vornehmen. Beispielsweise werden dort die Startadresse und Grösse des Memorys, die Position des Stack- und Base-Pointer und die anzuzeigenden Register und Flags definiert.

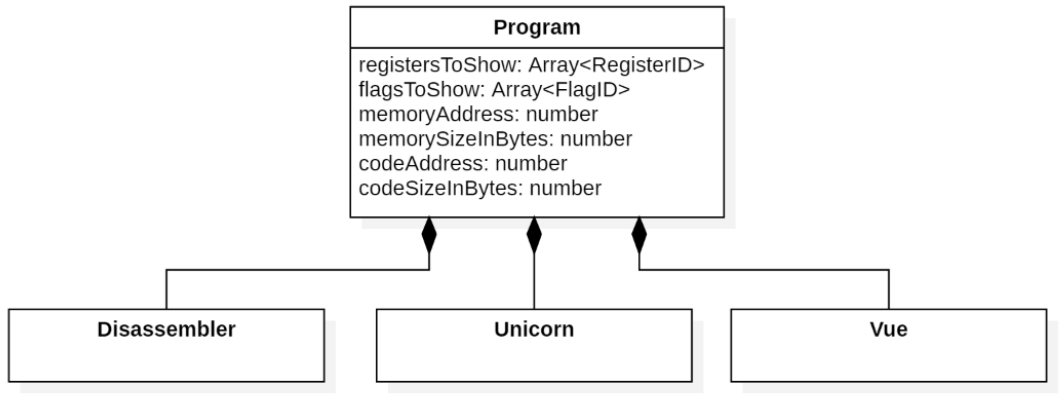


Abbildung C 5 Das Program Objekt

Controller

Der Controller beinhaltet die Klasse StepController. Diese ist dafür verantwortlich, in welchem Schritt, welche Daten im GUI aktualisiert werden und dass die entsprechenden Zustandsänderungen vom AnimationController animiert werden. Dafür erhält er bei der Konstruktion von der Simulator-Vue-Komponente das Program Objekt, das eine Unicorn- und Capstone-Instanz beinhaltet. Ausserdem erstellt er mithilfe des fillDataService das State-Objekt mit den Daten aus dem Emulator, das dann von der Simulator-Vue-Komponente abgefragt werden kann.

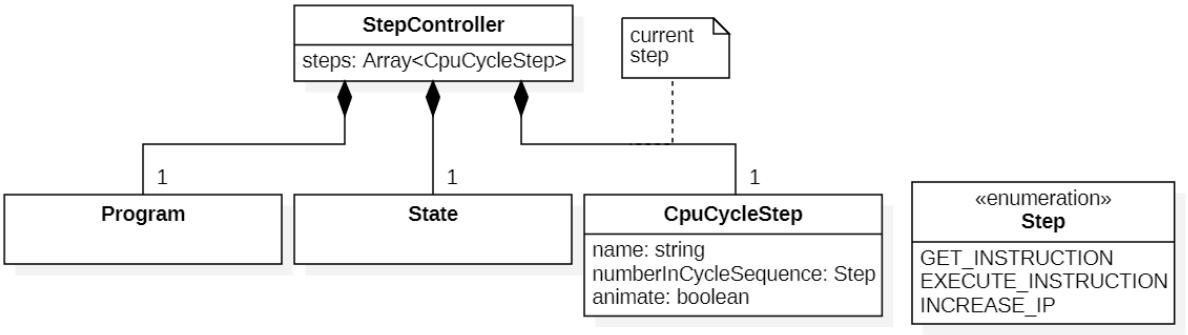


Abbildung C 6 Der Controller

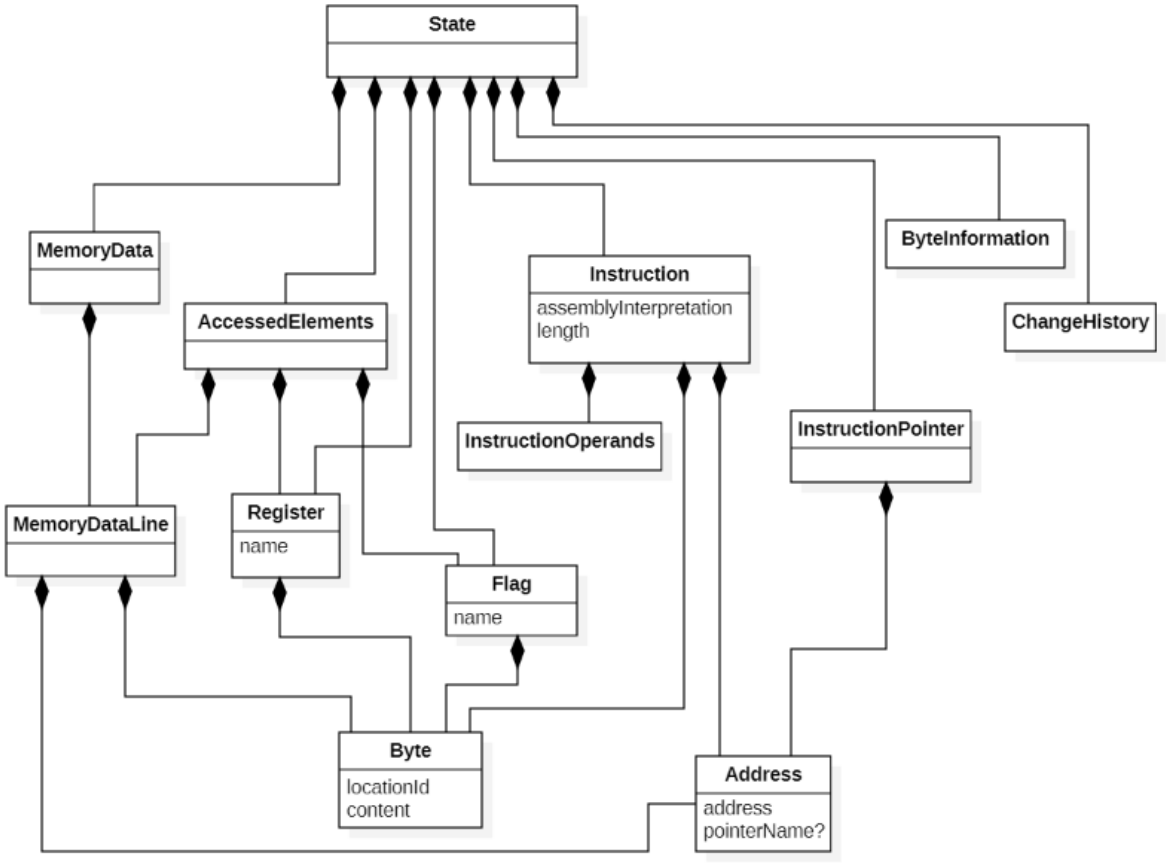


Abbildung C 7 Das State Objekt

Der Controller beinhaltet ein Array steps, das die Schritte des Prozessor-Zyklus beinhaltet. Er bietet die Funktion “nextStep” an, die einen Schritt vorwärts geht und die zum aktuellen Schritt gehörenden Zustandsänderungen am State-Objekt und die Animationen im AnimationController startet.

Im Schritt “Get Instruction” holt der Controller die neue Instruktion vom Instruction Service und startet die entsprechende Animation im AnimationController. Danach führt er die Instruktion aus und passt mithilfe der Data Services das AccessedElements Objekt im State an. Die Instruktion wird jedoch noch nicht animiert. Dies geschieht erst im Schritt “Execute Instruction”, wo der Animation Controller mithilfe der AccessedElements alle Zugriffe animiert und aktualisiert. Im Schritt “Increment Instruction Pointer” wird dann der Instruction Pointer analog zum ersten Schritt aktualisiert und die Animation gestartet.

DataService

Die DataService dienen dazu das State-Objekt mit Daten aus dem Emulator abzufüllen, damit diese dann im Simulator dargestellt werden. Beim Start des Simulators werden sie vom FillDataService aufgerufen, um das State-Objekt zu initialisieren. Danach werden sie vom Controller und vom Animation Controller aufgerufen, um die im aktuellen geänderten Daten vom Emulator abzufragen und im State Objekt anzupassen. Beispielsweise liest der InstructionSerivce die Bytes der aktuellen Instruktion aus dem Memory des Emulators und übergibt diese dann an Capstone, um das Instruction-Objekt zu erstellen.

In der folgenden Abbildung werden die Beziehungen innerhalb der DataService Komponente aufgezeigt. Ausserdem ist ersichtlich, welche Klassen auf die Komponente Emulator zugreifen. Die Zugriffe auf die Komponenten Interfaces und Helper werden nicht dargestellt.

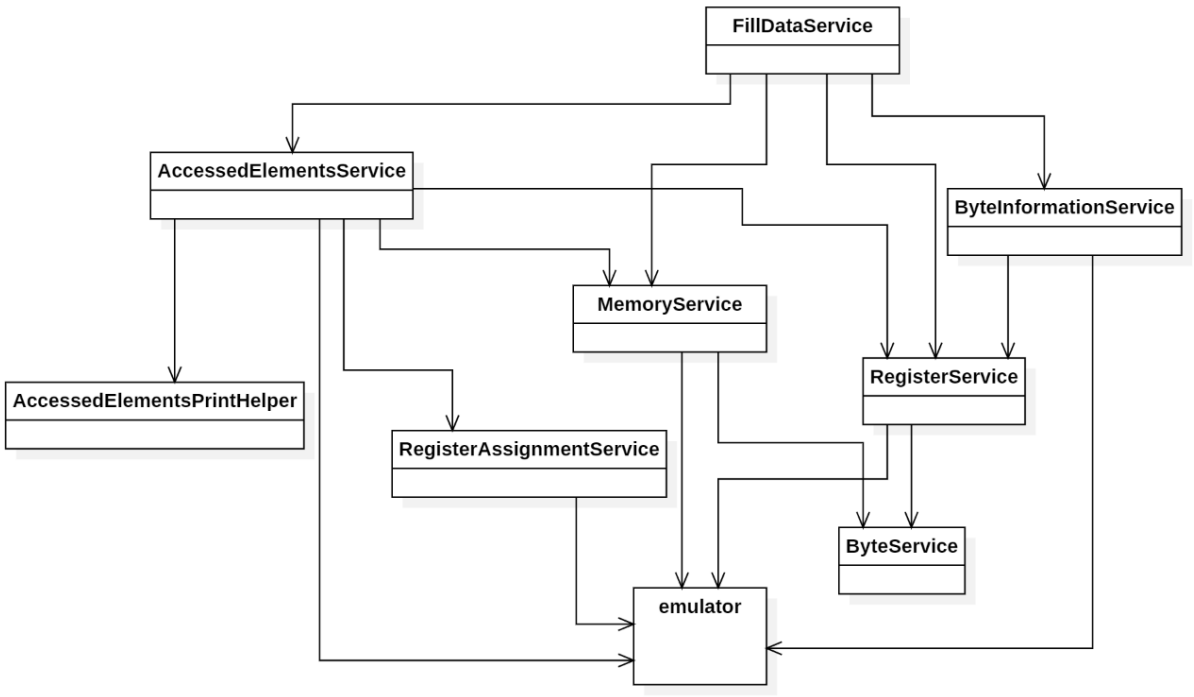


Abbildung C 8 Die DataService Komponente

RegisterAssginmentService

Wir überprüfen im Schritt “Get Instruction” anhand der InstructionOperands, ob die Register, auf die zugegriffen wird, bereits im Array RegistersToShow im Program Objekt vorhanden sind. Falls nicht wird die betreffende RegisterID hinzugefügt. Danach werden die Register aktualisiert und im Simulator angezeigt. Damit beim Öffnen des Simulators bereits Register dargestellt werden, füllen wir beim Starten des Simulators bereits die Register RAX und RBX in den RegistersToShow Array des Program Objekts.

Nun wollten wir natürlich verhindern, dass wenn eine Instruktion das Unter-Register EAX aufruft, dieses hinzugefügt wird, obwohl RAX, das EAX beinhaltet, bereits angezeigt wird. Ausserdem wollten wir, dass bei den Animationen die Unterregister zum richtigen Long-Size-Register fliegen. Deshalb haben wir einen RegisterAssignmentService hinzugefügt. Dort machen wir ein Mapping von RegisterIDs der Unterregister zur RegisterID des Long-Size-Register, die sie beinhalten. Wir haben dort alle Register zugewiesen, die in der Bsys1 Vorlesung angegeben wurden[25].

[25] S. Richter, Vorlesung Betriebssysteme 1: Intel 64 Architektur und C-Toolchain, Rapperswil: OST Ostschweizer Fachhochschule, 2020, pp.12.

Vor dem Hinzufügen der Register zum RegistersToShow Array wird die Funktion getLongSizeRegister vom RegisterAssignmentService aufgerufen, der das Long-Size-Register aus der Map zurückgibt. Falls das Register nicht gefunden wird, gibt die Funktion wieder diejenige Register Id zurück, die sie als Parameter erhalten hat. Um zu verhindern, dass das Register EFLAGS hinzugefügt wird, wenn die Flags modifiziert werden, füllen wir dieses bei den Registerzugriffen im instructionOperandsService nicht in die Read-/WriteAccess Arrays ab.

Bei den Animationen benötigen wir die LocationIds derjenigen Bytes in den Registern, die in die Animation involviert sind. Dafür prüfen wir jeweils, ob es sich bei dem zu animierenden Register, um ein Int-Size-, Short-Size, Low-Char- oder High-Char Register handelt. Dies machen wir anhand der Grösse des Registers und bei High- und Low-Char zusätzlich anhand des letzten Buchstabens im Registernamen. Falls es sich um eines dieser Unterregister handelt, geben wir diejenigen LocationIds aus dem Long-Size-Register zurück, wo das Unterregister liegt. So fliegen zum Beispiel die Bytes des Registers EAX in die erste Hälfte des Registers RAX.

AccessedElements Objekt

Im InstructionOperands Objekt aus der Disassembler Konstante erhalten wir nur den Namen und die Adressen der Elemente, auf die die CPU während der Ausführung zugreift. Für die Animationen der betroffenen Elemente, benötigen wir die richtigen Inhalte, die gelesen und geschrieben werden. Deshalb haben wir dem State ein Attribute “currentAccessedElements” hinzugefügt. Im Schritt “Get Instruction” wird im Controller bereits die Instruktion ausgeführt, die dann im Schritt “Execute Instruction” animiert werden soll. Bevor wir die Instruktion ausführen, lesen wir die Werte aus den Registern, die im InstructionOperands im “registerRead” angegeben sind und füllen die Werte ins Objekt AccessedElements ab. Ausserdem füllen wir die Immediate in eine MemoryDataLine ab, um diese im Simulator mit ByteVue Komponenten darzustellen.

Dann wird die Instruktion ausgeführt. Bei einem Zugriff aufs Memory greift der Memory Access Hook. Bei einem Read Access lesen wir aus dem Memory die Daten aus und erstellen eine MemoryDataLine mit der Adresse und den gelesenen Bytes, um diese später in der Animation anzuzeigen. Beim Write Zugriff erhalten wir den Wert, der geschrieben wird und füllen diesen in die MemoryDataLine ab. Anhand der Startadresse, die wir ebenfalls vom Hook erhalten, können wir den Bytes die richtige LocationId zuweisen, um sie im HTML-

Dokument zu identifizieren. Wenn die Bytes später animiert werden, kann der AnimationService, anhand der LocationId erkennen, wo das Byte hinfliegen muss.

Nach der Ausführung der Instruktion lesen wir die Register aus, die im InstructionOperands im “registerWrite” angegeben sind und erhalten so die Register für den registersWriteAccess Array im AccessedElements Objekt. Danach lesen wir die Flags aus dem Test und Write Access aus. Es gibt Instruktionen, die dieselben Flags testen und modifizieren. Diese werden im Simulator leider falsch animiert, denn Capstone liefert uns nur einen Zugriffsmodus. Wird ein Flag getestet und zusätzlich modifiziert erhält es den Modus “MOD” und nicht mehr “TEST”. Da die Flags mit Zugriffsmodus “TEST” deshalb auch nach der Ausführung der Instruktion denselben Wert haben, fragen wir diese, wie die Schreibzugriffe erst nach der Instruktions-Ausführung ab.

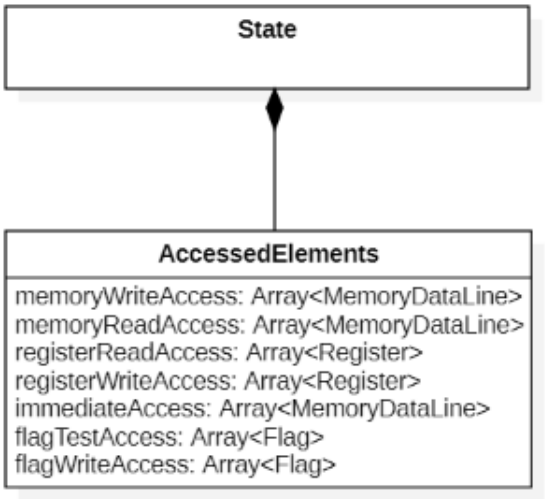


Abbildung C 9 Das AccessedElements Objekt

AccessedElementsPrintHelper

Für das Change Log wird die Information, was geändert wurde während einer Instruktion, in den Services dem State Objekt im Attribute ChangeHistory hinzugefügt. Bei einer Änderung an diesem Attribut, aktualisiert Vue.js den Inhalt der Change Log Komponente.

Das Attribut ChangeHistory im State Objekt ist ein Array aus ChangeHistory Objekten. Nach der Ausführung der Execute Instruction Animationen wird das neue ChangeHistory Element zum changeHistory Attribute im State hinzugefügt. Dafür werden die Write-Zugriffe aus den accessedElements des State in Strings umgewandelt und zum changedElements Attribute

des neuen ChangeHistory Objekt hinzugefügt. Dann wird das Instruction Attribute auf die Assembly Interpretation der Instruktion gesetzt.

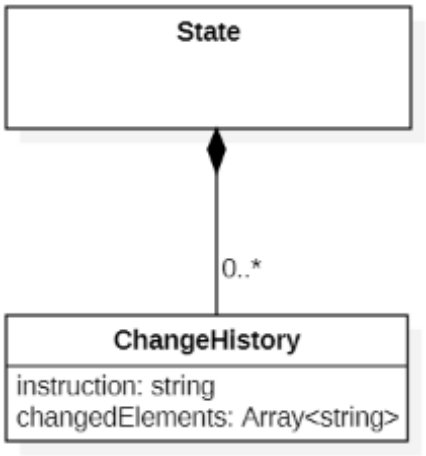


Abbildung C 10 Das ChangeHistory Objekt

ByteInformation Objekt

Das ByteInformation Objekt ist ein Attribute des State Objekts. Es dient dazu, die Information ans GUI weiterzugeben, welche Bytes benutzt wurden und eingeblendet werden sollen und welche Bytes zum Stack-, Base- oder Instruction-Pointer gehören.

Die Informationen im ByteInformation Objekt werden während den Animationen aktualisiert. Bei der Ausführung der Instruktion werden jeweils die Stack- und Base-Pointer-Register ausgelesen, die acht dazugehörigen Bytes bestimmt und die Informationen in den beiden PointerInformation Attributen im ByteInformation Objekt abgespeichert. Beim Erhöhen des Instruction-Pointers und beim Auslesen der Instruktion wird dasselbe mit dem Instruction-Pointer und den dazugehörigen Instruction-Bytes gemacht. Sobald etwas im ByteInformation Objekt verändert wird, werden die Bytes von Vue.js im GUI aktualisiert.

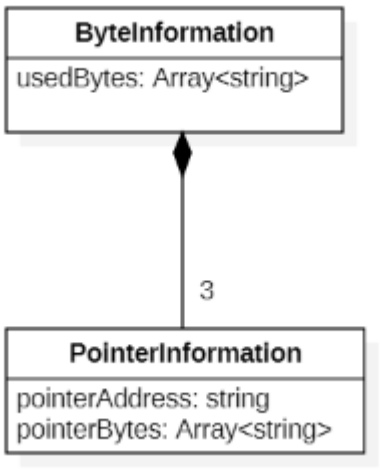


Abbildung C 11 Erste Umsetzung des ByteInformation Objekts

Zu Beginn haben wir in das usedBytes Array des ByteInformation Objekts alle LocationId Strings derjenigen Bytes, die als benutzt angezeigt werden sollen, abgefüllt. Es wurden alle Bytes im Codebereich und zusätzlich bei jedem Zugriff aufs Memory und jeder Veränderung der Pointer die entsprechenden Bytes hinzugefügt. Als wir dann das erste längere Programm gestartet haben, haben wir bemerkt, dass sich die Performance extrem verschlechtert hat. Grund dafür war, dass als erstes alle Bytes des langen Maschinencodes in den usedBytes Array abgefüllt wurden und dann bei jeder Aktualisierung pro Byte der grosse Array durchgetestet werden musste.

Um das Problem zu beheben haben wir dem ByteInformation Objekt ein Attribut CodeInformation hinzugefügt, das die Startadresse und die Grösse des Codes angibt. Ausserdem haben wir die LocationIds statt als Strings, als Numbers hinzugefügt. So kann die ByteVue Komponente anhand der LocationId mit zwei Vergleichen entscheiden, ob sie im Codebereich liegt oder nicht, statt sie immer mit allen usedBytes abzugleichen. Somit müssen die Bytes aus dem Codebereich nicht mehr zu den usedBytes hinzugefügt werden. Durch den kleineren UsedBytes Array war die Performance wieder im selben Bereich, wie vor der Einführung des ByteInformation Objekts.

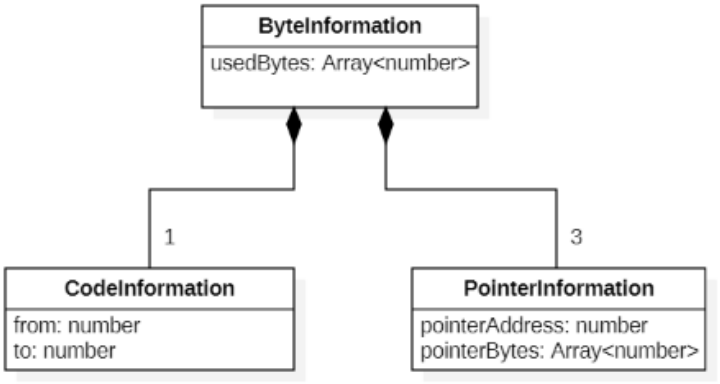


Abbildung C 12 Zweite Umsetzung des ByteInformation Objekts, die das Performance Problem löste

AnimationService

Der AnimationController wird vom Controller aufgerufen und steuert die Abläufe der Animationen. Im AnimationService werden die Move und CPU-Box Animationen gestartet. In AttentionAnimation werden jeweils die pulsierenden Animationen, die im GUI signalisieren, welche Elemente gerade betroffen sind, gestartet. Ausserdem werden die jeweiligen Elemente mithilfe des ScrollHelper in die Ansicht gescrollt.

Damit der AnimationController weiss, welche Daten er im Schritt “Execute Instruction” animieren muss, erhält er im State Objekt das Objekt AccessedElements mit den entsprechenden Registern, Memory Lines, Flags und Immediates. Mithilfe von ShowExecutionZoneBytes zeigt der AnimationController die Elemente, auf die zugegriffen wird, in der CPU-Box an. Nach der Animation eines Zugriffes werden die Daten im State Objekt aktualisiert. Um bei den Registern jedes einzelne zu aktualisieren, enthält das Program Objekt die Vue Instanz des Simulators, auf der die Änderung gemacht wird. Eine Änderung im Register Array auf dem State Objekt würde sonst von Vue nicht erkannt.

In der folgenden Abbildung werden die Beziehungen innerhalb der AnimationService Komponente aufgezeigt. Ausserdem ist ersichtlich, welche Klassen auf die Komponenten Emulator und DataServices zugreifen. Die Zugriffe auf die Komponenten Interfaces und Helper werden nicht dargestellt.

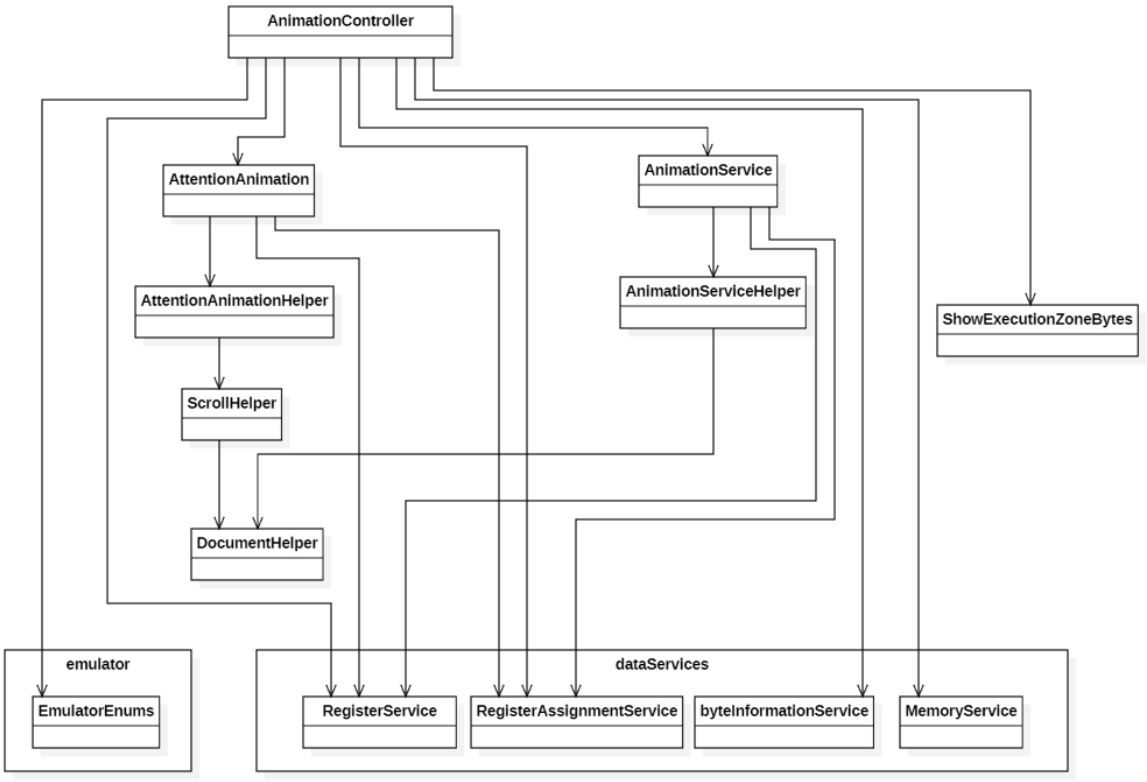


Abbildung C 13 Die AnimationService Komponente

Funktionsweise Move-Animation

Vor dem Move wird eine Attention Animation am Start und am Ziel ausgeführt, damit man sieht von wo, nach wo die Bytes fliegen werden. Dies wird vom AttentionAnimation Service erledigt. Danach wird die Move Animation vom Animation Service gestartet. Dafür werden die LocationIds der Bytes ausgelesen, die animiert werden sollen. Diese dienen als eindeutige Identifizierung im HTML. Die Ziele der Animation können mithilfe des HtmlIdService erstellt werden. Beispielsweise hängt dieser beim Lesen von Daten den Prefix 'animate_read_' an die LocationIds. Die CPU-Box verwendet denselben Service und zeigt somit Bytes mit diesen LocationIds an. Somit werden die Bytes dann vom Animation Service gefunden.

Die Funktion moveByte im AnimationServiceHelper sucht die angegebenen Ids im HTML Dokument und findet somit die betroffenen Start und Ziel-Elemente. Das Byte an der Startposition wird geklont und die Parameter für die Verschiebung so gesetzt, dass das Byte zum Element an der Endposition fliegt. Danach wird auf dem geklonten Byte die CSS-Klasse “moveByte” gesetzt. Dies triggert die Animation, die in der AnimationHelper Vue-Kompo-

nente definiert ist. Das Byte wird dann verschoben. Sobald die Animation zu Ende ist, werdem die geclonten Bytes wieder entfernt und der AnimationController aktualisiert den State, damit es so aussieht, als ob die Bytes im GUI durch die geklonten ersetzt werden.

AttentionAnimation Typen

Es gibt drei Typen von AttentionAnimations: Attention, AttentionPulse und Pulse. Diese sind alle in der Vue-Komponente AnimationHellper implementiert. Sie werden ausgeführt, wenn man die gleichnamige CSS-Klasse im AttentionAnimationHelper setzt.

- Attention: Dieser Typ lässt das Element kurz grösser werden und abdrehen.
- AttentionPulse: Dieser Typ führt dieselbe Animation wie Attention aus und lässt zusätzlich das Element stark pulsieren, bzw. zeigt einen Kreis um das Element, der grösser wird (siehe Abbildung).
- Pulse: Dieser Typ lässt das Element kurz pulsieren, bzw. zeigt einen kleineren Kreis als die AttentionPulse Animation, lässt das Element jedoch nicht rotieren.

Die AttentionPulse Animation wird vor der Move-Animation eines Bytes verwendet, um die Aufmerksamkeit des Benutzenden zu erlangen. Durch den grossen Puls Effekt bemerkt er die Animation, auch wenn er auf eine andere Stelle des Bildschirms fokussiert ist. Die Pulse Animation lässt im Gegensatz zur AttentionPulse die Elemente weniger pulsieren und dient dazu ein Zielpunkt für die Move-Animation oder ein aktuell betroffenes Element kurz anzudeuten.

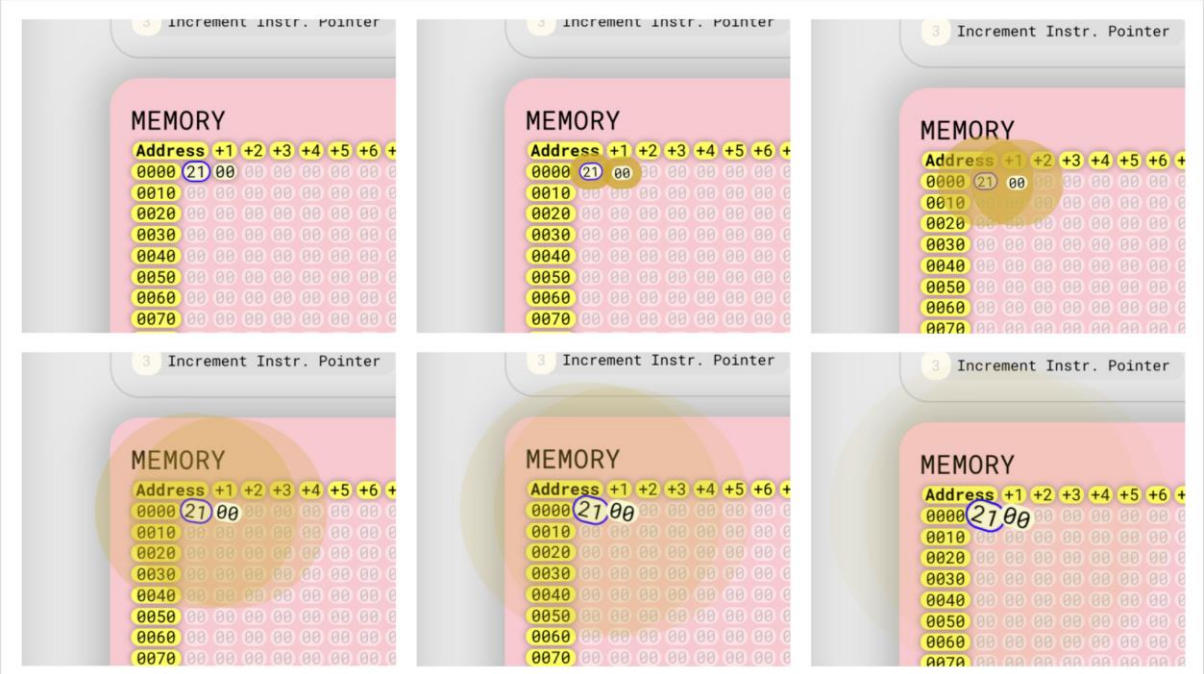


Abbildung C 14 Ablauf der AttentionPulse Animation

Interfaces und Helper

Die Interfaces beschreiben die Objekt-Strukturen, die im Simulator verwendet werden. Zum Beispiel besteht das State Objekt aus verschiedenen Interfaces (siehe Abbildung C 7 Das State Objekt). Die Entscheidung, dass wir Interfaces statt Klassen verwenden, haben wir in der Studienarbeit getroffen und deshalb so weitergeführt.

Die Interfaces werden von verschiedenen Architekturschichten verwendet. Ebenso die Helper Klassen. Dort gibt es den UInt8ArrayHelper, der UInt8Arrays in verschiedene Array Typen umwandelt. Ausserdem gibt es den htmlIdService, der dafür sorgt, dass die Vue Komponenten und die Services dieselben htmlId verwenden. Zum Beispiel kann der AnimationService bei der Move-Animation anhand der htmlId die Position des HTML-Elements erhalten. Damit diese Id's immer gleich aufgebaut sind und auch an einem Ort geändert werden können, werden sie im Objekt HtmlId definiert. Ein Byte, das in der Current Instruction angezeigt wird, erhält zum Beispiel den Prefix "CurrInstr" in seiner HTML-Id. Um die Id zu erhalten kann man die Funktion getLocationIdsForCurrentInstruction aus dem HtmlIdService für eine Instruktion alle LocationIds generieren lassen.

Die HtmlIds werden auch in den Vue-Komponenten verwendet. Bei den CSS-Klassen, die dynamisch während den Animationen gesetzt werden, funktioniert das leider nicht. Deshalb werden diese zwar im Objekt htmlClass in HtmlId.ts definiert, können jedoch nur von den Services verwendet werden. Wenn man ihren Namen ändern würde, müsste man dies auch in den CSS-Klassen Selektoren in den Vue-Komponenten anpassen.

Disassembler

Der DisassemblerService bearbeitet den Zugriff auf die Capstone Library. Er erhält von dort die Informationen zu den Operanden, die dann im InstructionOperandsService ins InstructionOperands-Objekt abgefüllt werden.

Diesen Wrapper hat eine sehr ungewöhnliche Art die Instruktionsdaten aus der C Umgebung zu lesen. Er greift direkt via Pointer und Offsets auf ein C Struct zu. Die Offsets entsprechen jeweils den aufsummierten Grössen der Datentypen im Struct. Da es funktionierte, änderten wir daran nichts. Jedoch mussten wir an die Instruktionsdetails gelangen, die nur über eine Referenz erreichbar waren. Die Instruktionsdetails sind ebenfalls in einem Struct gespeichert, jedoch braucht es sehr viel Logik, um die Informationen auszulesen, da jeweils

nicht alle Werte genutzt werden und man einige Überprüfungen auf den Nullwert machen müsste. Wir waren etwas verunsichert, da wir den Aufwand, die Informationen korrekt aus dem C Kontext zu erhalten, als hoch einschätzen.

In der folgenden Abbildung werden die Beziehungen innerhalb der Disassembler Komponente aufgezeigt. Ausserdem ist ersichtlich, welche Klassen auf die Komponenten Emulator, DataServices, Interfaces, Nasm und Helper zugreifen. Ausserdem greift die Klasse DisassemblerService auf die Capstone Library zu.

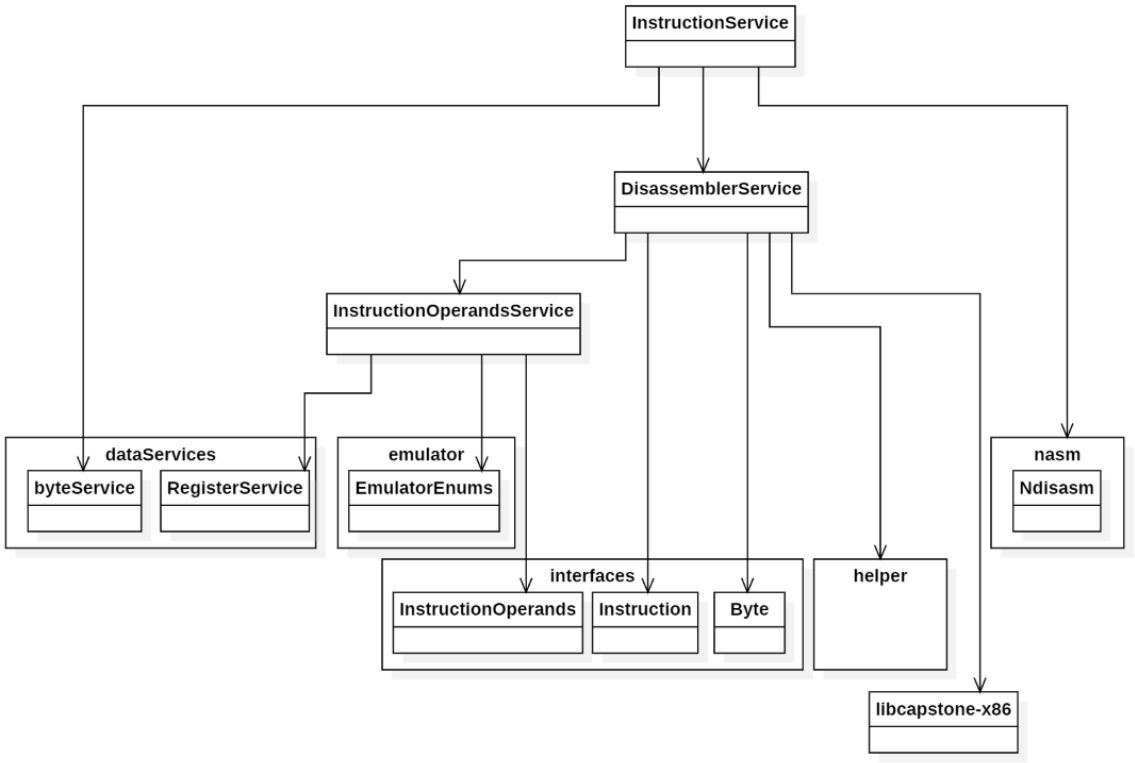


Abbildung C 15 Die Disassembler Komponente

InstructionOperands Objekt

Von Capstone erhalten wir nun die Informationen zur Instruktion und zu den Operanden. Dort werden auch Zugriffe auf die Register und Flags erkannt, die nicht in den Operanden vorkommen, zum Beispiel das RSP Register bei “push”. Wir wissen so zwar nicht, was in die Register geschrieben wird, wir können diese jedoch vor und nach der Ausführung einer Instruktion auslesen und so die Werte bestimmen.

Beim Memory Zugriff erhalten wir von Capstone nur Informationen, wenn es sich dabei um einen Operanden handelt und wir erhalten die Adresse, auf die zugegriffen wird, nicht. Deshalb nutzen wir dort einen Memory Access Hook, den Unicorn anbietet. Wir erhalten in der

Callback-Funktion die Memory Adresse, auf die zugegriffen wurde und wie viele Bytes betroffen sind. Bei einem Schreibzugriff erhalten wir zusätzlich den Wert, der geschrieben wird. Beim Read können wir mit den gegebenen Informationen auf das Memory zugreifen und den Wert auslesen.

Der von Capstone erhaltene String im JSON Format übergeben wir dem instructionOperandsService, der ein InstructionOperands Objekt erstellt, dass wir dem Instruction Objekt hinzufügen. Die folgenden Bilder zeigen Beispiele für den String, den wir von Capstone für die jeweiligen Instruktionen erhalten. Bei den Operanden gibt es drei Typen “MEM” (Memory), “IMM” (Immediate) und “REG” (Register). Die Immediate ist eine Art Konstante, die man bei gewissen Instruktionen als Operand angeben kann.

add ebx, [rbp + 0x10]


```
{
    "Prefix": "0x00 0x00 0x00 0x00 ",
    "Opcode": "0x03 0x00 0x00 0x00 ",
    "rex": "0x0",
    "addr_size": "8",
    "modrm": {
        "modrm_value": "0x5d",
        "modrm_offset": "0x1"
    },
    "disp": {
        "disp_value": "0x10",
        "disp_offset": "0x2",
        "disp_size": "0x1"
    },
    "sib": { "sib_value": "0x0" },
    "op_count": "2",
    "operands": [
        {"type": "REG", "value": "ebx", "size": "4", "access": "READ_WRITE" },
        {"type": "MEM", "reg_base": "rbp", "disp": "0x10", "size": "4", "access": "READ"}],
    "registers_read": [ "ebx", "rbp"],
    "registers_modified": [ "rflags", "ebx"],
    "EFLAGS": [
        { "name": "AF", "access": "MOD" },
        { "name": "CF", "access": "MOD" },
        { "name": "SF", "access": "MOD" },
        { "name": "ZF", "access": "MOD" },
        { "name": "PF", "access": "MOD" },
        { "name": "OF", "access": "MOD" }]
}
```


Abbildung C 16 Output Capstone für "add" Instruktion

push qword [rax * 8]


```
{
    "Prefix": "0x00 0x00 0x00 0x00 ",
    "Opcode": "0xff 0x00 0x00 0x00 ",
    "rex": "0x0",
    "addr_size": "8",
    "modrm": {
        "modrm_value": "0x34",
        "modrm_offset": "0x1"
    },
    "disp": {
        "disp_value": "0x0",
        "disp_offset": "0x3",
        "disp_size": "0x4"
    },
    "sib": {
        "sib_value": "0xc5",
        "sib_index": "rax",
        "sib_scale": "8"
    },
    "op_count": "1",
    "operands": [
        {"type": "MEM", "reg_index": "rax", "scale": "8", "size": "8", "access": "READ" }],
    "registers_read": [ "rsp", "rax"],
    "registers_modified": [ "rsp"]
}
```


Abbildung C 17 Output Capstone für "push" Instruktion

jnz 0xc


```
{
    "Prefix": "0x00 0x00 0x00 0x00 ",
    "Opcode": "0x75 0x00 0x00 0x00 ",
    "rex": "0x0",
    "addr_size": "8",
    "modrm": { "modrm_value": "0x0" },
    "disp": { "disp_value": "0x0" },
    "sib": { "sib_value": "0x0" },
    "imm_count": "1",
    "imms": [
        { "imm": "0xc", "imm_offset": "0x1", "imm_size": "0x1" } ] ,
    "op_count": "1",
    "operands": [
        {"type": "IMM", "value": "0xc", "size": "8" }],
    "registers_read": [ "rflags"],
    "EFLAGS": [
        { "name": "ZF", "access": "TEST" }]
}
```


Abbildung C 18 Output Capstone für "jnz" Instruktion

Die Memory Operanden verwenden wir nicht, da wir den Memory Zugriff mit dem Memory Access Hook von Unicorn abfangen. Wir füllen diese Operanden der Vollständigkeit halber trotzdem ins InstructionOperands Objekt ab. Wir erhalten unteranderem die Grösse und den Zugriffsmodus auf das Memory. Es gibt dabei drei Modi: READ, WRITE, READ_WRITE. Wir erhalten ausserdem Informationen darüber, ob mithilfe eines Registers auf das Memory zugegriffen wird. Bei der Instruktion “add ebx, [rbp + 0x10]” erhalten wir mit dem Attribute “reg_base” den Namen des Registers RBP und unter “disp” den Wert, um den der Pointer verschoben wird, also “0x10”. Bei “push qword [rax * 8]” erhalten wir unter “reg_index” den Namen des Registers und bei “scale” den Wert, um den multipliziert wird, also “8”.

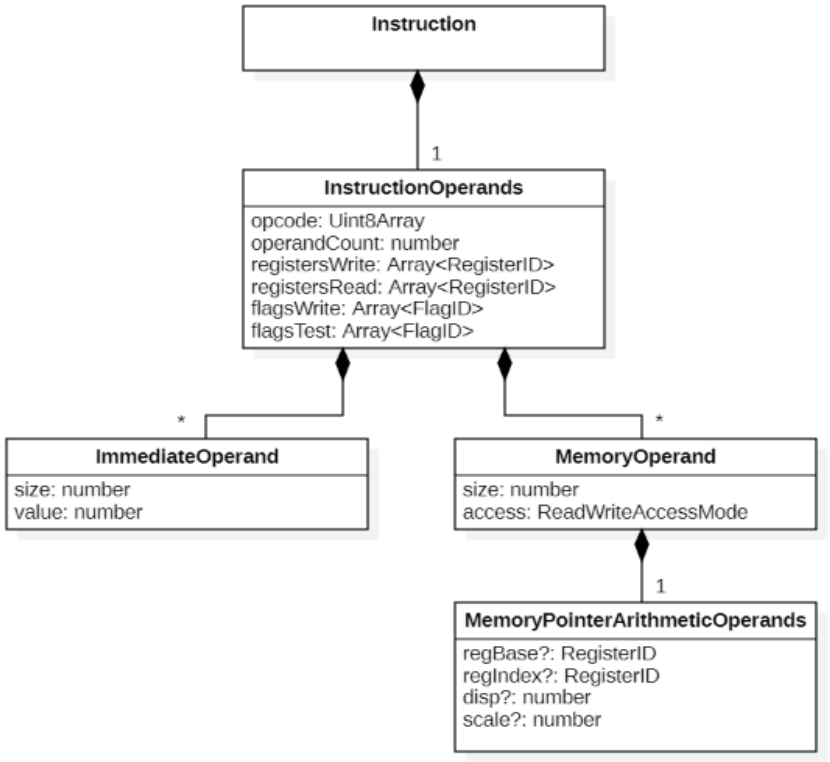


Abbildung C 19 Das InstructionOperands Objekt

Bei den Registern erhalten wir den Namen im Attribute “value” und die Information, wie auf die Registern in den Operanden zugegriffen wird. Wir suchen anhand des Namens die zugehörige RegisterID und fügen diese dann je nach Zugriffsmodus dem registersRead und/oder dem registerWrite Array in den Attributen des InstructionOperands hinzu. Neben den Operanden erhalten wir im JSON Objekt auch die Arrays “registers_read” und “registers_modified”. Dort sind neben den Registern aus den Operanden (Type Register und Type Memory) auch alle anderen Register aufgeführt, auf die zugegriffen wird, zum Beispiel RSP bei “push”. Wir suchen dann für Register aus diesen beiden Arrays die zugehörigen RegisterID.

Dann fügen wir sie zu den registersRead und/oder dem registerWrite Array in den Attributen des InstructionOperands hinzu, sofern sie nicht bereits enthalten sind. Die Register werden in den Arrays vom JSON Objekt nur jeweils einmal aufgeführt, auch wenn es zweimal gelesen wird. Deshalb füllen wir bewusst zuerst die Register Operanden ab, da zum Beispiel bei “xor rbx, rbx” RBX zweimal bei den Operanden angegeben wird.

Die Flags füllen wir ähnlich wie die Register ab. Wir suchen anhand des Flag namens die FlagID. Danach füllen wir es je nach FlagAccessMode in die FlagsWrite oder FlagsTest Array ab. Ein Flag erhält von Capstone einer der folgenden Modi: “UNDEF”, “MOD”, “TEST”, “RESET”, “PRIOR”, “SET”. Wir interpretieren “MOD”, “SET” und “RESET” als FlagWriteAccess und “TEST” als FlagTestAccess. FlagTestAccess wird dann wie die anderen Lesezugriffe animiert.

Emulator

Um auf die Unicorn Library zuzugreifen, wird der EmulatorService verwendet. Über diesen kann man Daten aus dem Emulator abfragen, oder eine Instruktion ausführen. In den EmulatorEnums werden die RegisterIDs als Enum gespeichert. Ausserdem kann man über die Funktion registerSize die Grösse eines Registersabfragen.

In der folgenden Abbildung werden die Beziehungen innerhalb der Emulator Komponente aufgezeigt. Ausserdem ist ersichtlich, dass der EmulatorService auf die Interfaces Komponente und die Unicorn Library zugreift.

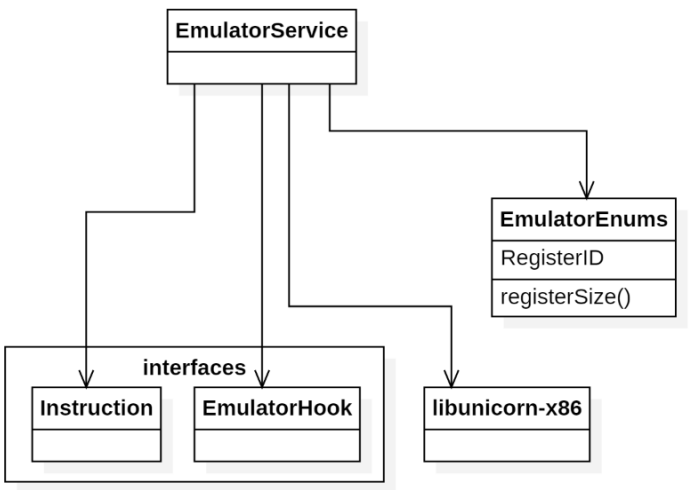


Abbildung C 20 Die Emulator Komponente

Nasm

In Nasm befinden sich die beiden Nasm und Ndisasm Services, durch die man auf die jeweiligen Libraries zugreifen kann.

Umsetzung Flags

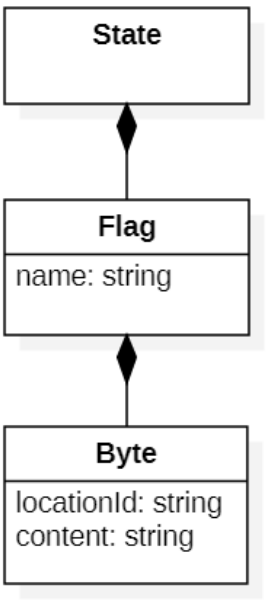
Die Status-Flags entsprechen den folgenden Bits im EFLAGS Register[26]:

- CF (Bit 0) Carry Flag
- ZF (Bit 6) Zero Flag
- SF (Bit 7) Sign Flag
- OF (Bit 11) Overflow Flag

Die Daten für die Flags speichern wir im Flag Objekt ab. Dieses beinhaltet einen Namen und im Attribut Content ein Byte Objekt. Um die Flag Werte im Byte Objekt darzustellen, haben wir den ByteService so angepasst, dass er nicht nur zweistellige Hexadezimal Zahlen akzeptiert, sondern auch eine einzelne eins oder null.

Analog zu den Registern gibt es im Objekt Program einen Array FlagsToShow, der im startSimulatorService mit FlagIDs abgefüllt wird. FlagID ist ein Enum der als Namen die Flag Abkürzungen und als Werte, deren Position im EFLAGS Register beinhaltet. Das SF ist zum Beispiel das siebte Bit im EFLAGS Register. Die Funktion getFlags im RegisterService fragt jeweils das EFLAGS Register im Simulator ab und erstellt dann die Flags des flagsToShow Array mit den entsprechenden Werten. Als Namen werden die Namen der Werte im Enum, zum Beispiel SF, verwendet. Die erstellten Flags werden dem State Objekt hinzugefügt und dann so an die Flag Vue-Komponente im Simulator weitergegeben. Möchte man ein weiteres Flag im Simulator anzeigen, ergänzt man seine Position im Enum FlagID und fügt den Wert im flagsToShow Array ein.

[26] Intel Corporation, «Status Flags,» in Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, 2021, pp. 3 - 16 Vol. 1.




```
export enum FlagID {
  CF = 0,
  ZF = 6,
  SF = 7,
  OF = 11,
}
```

Abbildung C 21 Das Flag Objekt und das FlagID Enum

Umsetzung Loading Spinner

Den Spinner startet man, indem man einen Boolean auf dem Button setzt. Dies machen wir jeweils zu Beginn der Funktion, die aufgerufen wird, wenn der entsprechende Button geklickt wird. Beim Routing funktioniert das nicht richtig, wahrscheinlich weil der Compiler das Setzen des Attributs wegoptimiert. Beim Editor wird nur die assembleCode Methode aufgerufen und danach zum Simulator navigiert. Deshalb setzen wir zuerst ein Timeout, das parallel zur assembleCode Methode läuft und navigieren erst dann zum Simulator. Wir haben versucht die Zeit so zusetzen, dass man nicht zu lange künstlich warten muss, aber der Loading Spinner trotzdem ersichtlich ist.

Kurz vor dem Routing zum Simulator laufen die Animationen in Zeitlupe. Wenn der Spinner erst kurz vor diesem Zeitpunkt startet, erkennt man ihn nicht richtig. Deshalb haben wir die Zeitdauer des Timeouts so gewählt, dass mindestens ein kleiner Teil des Spinners ersichtlich ist. Im Simulator haben wir das ähnlich umgesetzt wie im Emulator. Die Zeitdauer ist dort kürzer (400 statt 540 Millisekunden), da die assembleCode Funktion da nicht ausgeführt werden muss. Mit dieser Lösung bemerkt der Benutzende die längere Wartezeit praktisch nicht, sieht aber dafür den Loading Spinner und weiss, dass etwas passiert.

Umsetzung Animationen überspringen

Für die Umsetzung der Funktionalität Animationen zu überspringen, haben wir dem Objekt CPUCycleStep einen Boolean “animate” hinzugefügt. Die StepController Klasse besitzt als Attribut einen Array steps, der jeweils ein CPUCycleStep Objekte pro Schritt beinhaltet. Um die Animationen für einen Schritt auszuschalten, kann man in der CPU Cycle Komponente einen Toggle Button umstellen. Dieser löst einen Event aus, bei dem der Simulator dem Step Controller mitteilt, dass er beim betroffenen Step im steps Array das Attribute animate umstellen muss. Der StepController kann anhand des Attributs im aktuellen Schritt dem AnimationController mitteilen, ob er diesen animieren soll. Der Animation Controller führt dann je nach dem die Animationen aus oder aktualisiert nur die Daten.

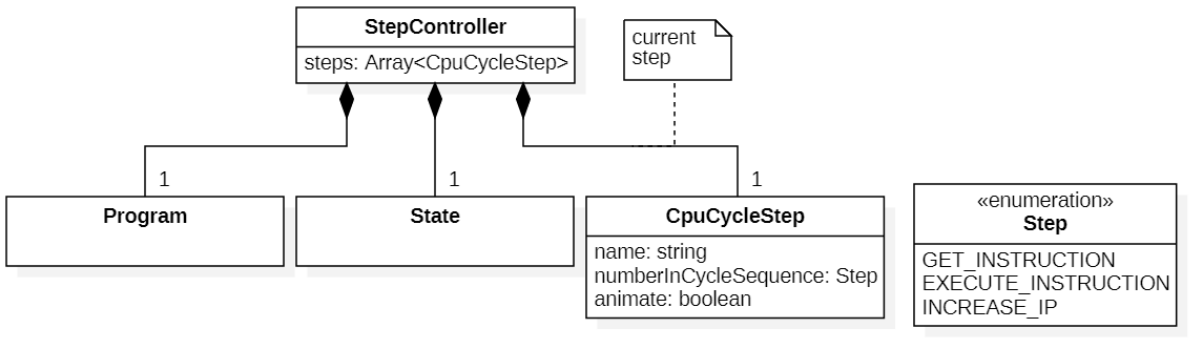


Abbildung C 22 Der Controller

Lizenz GPLv2 mit SPDX Identifier

Wie in der Arbeit erläutert hatten wir uns für die GPLv2 entschieden. Um es anderen leicht zu machen zu erkennen, dass man bei unserer GPLv2 Lizenz kein Upgrade auf die Version 3 machen kann, entschieden wir uns den SPDX Identifier[27] zu verwenden. Dieser wurde von der Linux Foundation entwickelt, um ein effizientes und zuverlässiges maschinelles Erkennen der Lizenzen zu ermöglichen. Die Nomenklatur haben sie unter dem Namen "Software Package Data Exchange® (SPDX®)" bei der ISO Kommission als Standard angemeldet[28].

[27] SPDX Workgroup, «SPDX License List,» 20 Mai 2021. [Online]. Available: https://spdx.org/licenses/ . [Zugriff am 15 Juni 2021].
[28] Kernel, «Linux kernel licensing rules», 13 Juni 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/process/license-rules.html#license-identifier-syntax . [Zugriff am 15 Juni 2021].

Alle unsere Files sind nun mit " SPDX-License-Identifier: GPL-2.0-only" als Kommentar geprägt.

```
/* SPDX-License-Identifier: GPL-2.0-only */
/*
* GCSx64 - x86_64 Graphical CPU Simulator
*
* Copyright © 2021 by Eliane Schmidli eliane.schmidli@gmail.com and Yves Boillat yvbo@protonmail.com
*
* This file is part of GCSx64 - x86_64 Graphical CPU Simulator
*
* GCSx64 is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, version 2 of the License only.
*
* GCSx64 is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with GCSx64.  If not, see https://www.gnu.org/licenses/ .
*/

<template>
  <div class="background">
```

Abbildung C 23 Lizenz mit SPDX-License-Identifier

Anhang D Konzept und Ergebnis Usability Test 1

Dokumentation Usability Test 1

Um bereits früh allfällige Fehler im Simulator zu erkennen, wurde in der Mitte des Projekts ein Usability Testing durchgeführt. Wir wollten sehen, wie die Benutzenden den Simulator verwenden und ob sie Schwierigkeiten mit der Bedienung haben. Ausserdem soll so die Verständlichkeit der Animationen und des Lerninhaltes sichergestellt werden. Es sollte anhand der Probleme klargestellt werden, welche Schwierigkeiten bestehen und welche Features höhere Priorität haben in der zweiten Hälfte des Projekts.

Wir haben für den zweiten Usability Test zum Abschluss dieser Arbeit geplant, als Testpersonen Studierende zu nehmen, die möglichst nahe an der Bsys1 Vorlesung sind, um zu sehen, ob unser Simulator ihrer Meinung nach zur Vorlesung passt. Um die Verständlichkeit des Prozessor-Zyklus im Simulator zu testen, haben wir uns entschieden Personen zu suchen, die die Bsys1 Vorlesung nicht kennen, oder sie nicht mehr so präsent haben.

Der Test wurde dreimal durchgeführt. Die Personen haben vor Ort oder via MS Teams mit Kamera und Screensharing an ihrem eigenen Laptop gearbeitet. Die erste Person studiert Wirtschaft und hat praktisch keine Informatikkenntnisse. Die anderen zwei sind Informatik Studierende, die das Modul Bsys1 vor zweieinhalb, bzw. dreieinhalb Jahren besucht haben und den Stoff deshalb nicht mehr ganz präsent haben. Die Testpersonen wurden nicht entschädigt. In diesem Dokument wird die Durchführung des Tests und die erhaltenen Ergebnisse dokumentiert. Eine genaue Auflistung der Erkenntnisse befindet sich im Anhang J und die dazugehörigen Protokolle sind im Anhang L aufgeführt.

Durchführung

Den Testpersonen wurde der Simulator als HTML File und ein File mit drei Beispielprogrammen ausgehändigt. Da es noch keine Beispielprogramm Buttons gibt, mussten die Testpersonen die Programme selbst hineinkopieren. Die Personen wurden gebeten, sich in die Übung im ersten Semester zu versetzen und bekamen den Auftrag, dass sie verstehen sollen, wie der Prozessor arbeitet. Ausserdem sollten sie laut denken und Fragen laut stellen, die jedoch erst ganz am Schluss von uns beantwortet wurden.

Im Vorinterview wurden bewusst nur nach der Arbeitsweise des Prozessors und nicht nach weiterer Theorie gefragt. Hätten wir eine Frage zum Stack oder Register gefragt, hätte es sein können, das wir den Probanden mit der Frage bereits einen Hinweis gegeben hätten, dass beispielsweise der Stack dargestellt wird.

Die Personen haben daraufhin den Simulator gestartet und das Beispielprogramm, das bereits im Simulator steht, ausgeführt. Da das Design des Simulators nicht Responsive war mussten einige der Probanden zuerst hinauszoomen, um den gesamten Simulator zu sehen. Die Bedienung des Simulators war aber sehr schnell klar. Schwierigkeiten bereitete jedoch die Navigation vom Simulator zurück zum Code Editor, da es dafür keinen Button gab. Keiner der Probanden hat bemerkt, dass man mit dem Zurück Knopf des Browser zurück gelangt wäre.

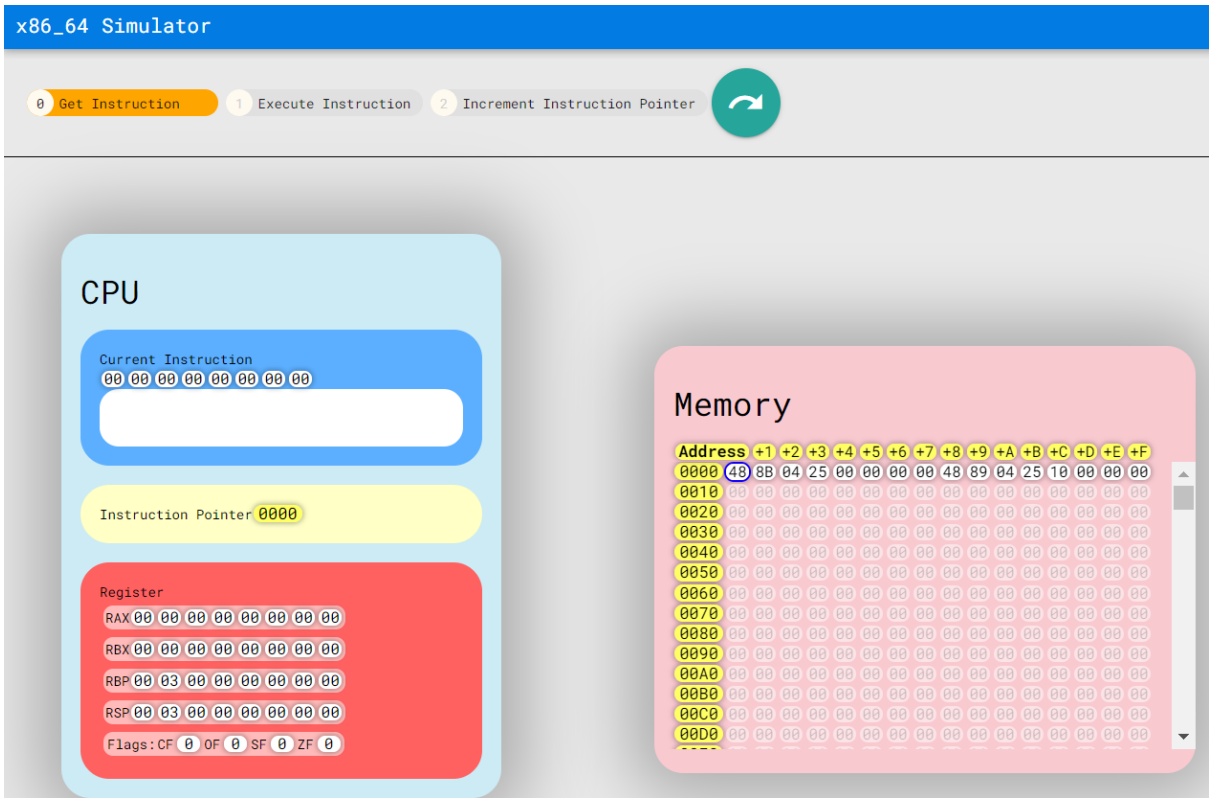


Abbildung D 1 Screenshot des Prozessor-Simulators zum Zeitpunkt des ersten Usability Tests

Falls die Personen es nicht selbständig gemacht haben, wurden sie nach einer gewissen Zeit aufgefordert, ein Beispielprogramm zu starten. Im Code Editor war es schwierig den Code auszuwählen und zu löschen, da man nur den Text im Textfeld und nichts anderes

markieren darf. Es ist ausserdem nicht möglich in das Textfeld zu schreiben, wenn man nicht daraufklickt. Bei zwei der Probanden sind beim Assemblieren Fehlermeldungen aufgetreten, die sie sofort erkannt und verstanden haben, obwohl die Fehlermeldung nicht sehr prominent dargestellt wurde.

Abbildung D 2 Screenshot des Code-Editors mit Fehlermeldung zum Zeitpunkt des ersten Usability Tests

Beispielprogramme

Die Beispielprogramme wurden bereits angeboten, um zu testen, ob diese helfen oder noch mehr verwirren. Dabei ist aufgefallen, dass die Titel sehr aussagekräftig sind und somit schon ein Gefühl geben, um was es im Programm geht. Zwei der Programme stammen aus den Übungen der Vorlesungen. Bei einem Funktionsaufruf wird immer ein Prolog ausgeführt und als erstes der Wert des Base Pointers auf den Stack gepushed (push rbp). Dies war auch Teil der Beispiel Programme. Die Probanden waren aber stark verwirrt. Da der Simula-

tor bereits benutzt wird, bevor der Stack erklärt wird, muss das aus den Programmen weggelassen werden. Dafür könnte man ein separates Programm schreiben, dass “Stack Programm” heisst.

```
SECTION .text
global main
main:
  push rbp
  mov rbp, rsp
```

Abbildung D 3 Prolog beim Funktionsaufruf

Danach wurden die Probanden aufgefordert, das Beispielprogramm mit der Multiplikation auszuführen. In diesem Programm gab es einen Loop und wir wollten erkennen, ob die Jump Instruktion klar ist. Die Animation war leider nicht verständlich, da der Instruction Pointer nicht bei Execute Instruction überschrieben wird. Es haben aber alle erkannt, dass der Pointer zurückgesetzt, und wieder dieselben Instruktionen ausgeführt werden.

Stack

Die Probanden, die Bsys1 besucht haben wurden aufgefordert ein Programm einzugeben, ohne dass gesagt wurde, dass es sich um ein Stack Programm handelt. Sie erkannten nach längerer Zeit, dass es sich bei den Registern RSP und RBP um Pointer handelt, die auf die farbigen Byte im Memory zeigen. Dass es sich dabei um den Stack Pointer handelt, hat nur eine Person erkannt und den Base Pointer keiner von beiden. Es braucht unbedingt noch Tooltips im Memory und eine spezielle Darstellung der Register RSP und RBP zum Beispiel analog zum Instruction Pointer.

Abbildung D 4 Darstellung des Stacks zum Zeitpunkt des ersten Usability Test

Person ohne Vorkenntnisse

Interessant war das Verhalten der Person, die noch keine Informatikkenntnisse hat, zu beobachten. Die Person hat zu Beginn anhand des Programms, das bereits im Code Editor steht, versucht, die Zusammenhänge der Zustandsänderungen zu erkennen. Da das Programm aus mov Instruktionen besteht, war der Opcode und die Länge der Funktionen identisch. Das war eher kontraproduktiv, da die Person dann die Increase Instruction Pointer Animation nicht richtig verstanden hat, da die Adresse einfach immer um 8 Byte hochgezählt wurde. Erst beim Multiplikationsprogramm war dann klar, warum der Pointer hochgezählt werden muss und dass er sich immer um die Länge der Instruktion erhöht. Deshalb muss im Default Programm noch eine Instruktion mit anderer Länge und Opcode vorhanden sein.

Der Proband ohne Vorkenntnisse hat nach zwei Durchführungen des Default Programms bereits den grössten Teil des Prozessorzyklus verstanden und hat bei ein paar Assembly Instruktionen erkannt, was sie machen. Dies spricht für unsere Animationen und unsere Abstraktionsebene, die wir gewählt haben. Den Stack hat er bis zum Schluss nicht verstanden.

Design CPU-Box und Change Log

Für die Entwicklung der nächsten Features haben wir bereits vor dem Test ein Designansatz ausgearbeitet und diesen dann den Testpersonen vorgestellt.

Die CPU-Box war eine Idee, um zu zeigen, dass die CPU etwas mit den gelesenen Daten macht. Die Probanden haben erkannt, dass es dabei um die CPU geht die Daten liest, verarbeitet und den Output schreibt. Das Design kam gut an, da somit die Daten beim Lesen nicht einfach zur CPU fliegen und verschwinden sondern ein Ziel haben. Ausserdem kann so eine Verwirrung zwischen Get Instruction und Lesen aus dem Memory verhindert werden.

Der Change Log ist eine Idee, um ein Textfeld einzublenden, das zeigt, welche Instruktionen ausgeführt wurden und welche Daten verändert wurden. Das kam auch sehr gut an. Zwei der Probanden haben sich einen “Einen Schritt zurück” Button gewünscht, mit dem Sie einen Schritt wiederholen könne. Sie fanden jedoch beide, dass das Change Log denselben Zweck erfülle. Ausserdem fanden sie den Vorschlag gut, dass man auf dieselbe Art den Code einblenden kann, den man im Editor eingegeben hat. So kann man verhindern, dass man den Code in einem andern Fenster noch parallel zum Simulator öffnen muss.

Fazit

Allen Probanden hat der Simulator gefallen und sie würden ihn in der Übung gerne verwenden. Wenn ein “Zurück zum Editor” Button hinzugefügt wird, ist die Bedienung innerhalb 5 Minuten sichergestellt. Das Verständnis über den Prozessorzyklus ist innerhalb 30 Minuten ebenfalls sichergestellt, sofern der Execute Instruction und Get Instruction Schritt noch besser unterscheidbar sind.

Ausgehändigte Beispielprogramme

Diese Programme wurden den Testpersonen des ersten Usability Tests ausgehändigt. Die Programme „Variable Switch“ und „Addition of two big numbers” stammen aus der Vorlesung Bsys1.

Variable Switch

```
BITS 64

SECTION .data
    firstVar:
        dq 0x1337
    otherVar:
        dq 0xcafe

SECTION .text
    global _start
    _start:
        mov rax , [ firstVar ]
        mov rbx , [ otherVar ]
        mov [ firstVar ], rbx
        mov [ otherVar ], rax
```

Simple Multiplication

```
BITS 64

SECTION .text
global main
main:
mov rsp, 0x300
push rbp
mov rbp, rsp
mov rax, 0x4E
mov rbx, 0x3
loop:
    add rcx, rax
    dec rbx
    cmp rbx, 0
    jne loop
mov rcx, [0x50]
xor rax, rax
xor rcx, rcx
pop rbp
ret
```

Addition of two big numbers

```
BITS 64

SECTION .data
; RESULT VARIABLE! DO NOT EDIT!
res: dq 0x0, 0x0, 0x0, 0x0

; INPUT VARIABLES. PUT YOU VALUES HERE!
op1: dq 0x3344556677889900, 0xaabbccddeeff1122, 0xf000
op2: dq 0xccbbaa9988776700, 0x554433221100eedd, 0xc

SECTION .text
global main
main:
  push rbp
  mov rbp, rsp

  mov rcx, 0

  mov rax, [op1 + 8 * rcx]
  add rax, [op2 + 8 * rcx]
  mov [res + 8 * rcx], rax

  inc rcx

  mov rax, [op1 + 8 * rcx]
  adc rax, [op2 + 8 * rcx]
  mov [res + 8 * rcx], rax

  inc rcx

  mov rax, [op1 + 8 * rcx]
  adc rax, [op2 + 8 * rcx]
  mov [res + 8 * rcx], rax

  xor rax, rax
  pop rbp
  ret
```

Anhang E Konzept und Ergebnis Usability Test 2

In diesem Dokument wird zuerst das Testkonzept des zweiten Usability Tests und danach die Auswertung der Durchführungen beschrieben. Zusätzlich haben wir uns bei den Verbesserungsvorschlägen Gedanken gemacht, wie diese in einer Folgearbeit umgesetzt werden können. Die genaue Auflistung des Feedbacks befindet sich im Anhang K und die dazugehörigen Protokolle sind im Anhang M aufgeführt.

Testkonzept Usability Test 2

Wissensziele

Für den zweiten Usability Test haben wir verschiedene Fragen definiert, die wir durch den Test beantwortet haben wollen.

Aus den Use Cases ist eine Anforderung, dass die Zustandsübergänge ersichtlich sind und dass die Hardware abstrakt verstanden wird. Wir verstehen darunter, dass man sieht welche Komponenten zur CPU gehören, und wie sie in den Prozessor-Zyklus involviert sind.

Da im letzten Usability Test die Darstellung des Stacks noch unklar war, und man sich einen Änderungsverlauf und Code anzeige gewünscht hat, wollten wir nun die Bedienbarkeit und Verständlichkeit unserer Umsetzung dieser Features testen.

Wir wollten wissen, ob wir unsere Ziele aus den nicht-funktionalen Anforderungen erreicht haben. Diese waren, dass die Benutzung innerhalb von 5 Minuten klar ist und dass man innert 30 Minuten, den Prozessor-Zyklus aus der Vorlesung versteht. Dass die Benutzung verständlich ist, haben wir uns so definiert, dass der User den Editor und anschliessend den Simulator starten kann, den nächsten Schritt ausführen kann und wieder zurück zum Editor gelangen kann.

Das Ziel der Arbeit ist einen Simulator zu entwickeln, der im Gegensatz zu anderen Tools besser für die Bsys1 Vorlesung geeignet ist. Wir mussten also herausfinden, ob unser Ergebnis verständlicher und ansprechender ist als die Konkurrenz Produkte. Ausserdem wollten wir Feedback zum Aussehen und Spassfaktor unseres Produktes. Für den Ausblick und für Folgearbeiten, wollten wir zudem abklären, was sich die Studierenden noch wünschen.

Testpersonen

Um herauszufinden, ob unsere Applikation den Erwartungen von Bsys1 Studierenden gerecht wird und auch den Vorlesungsstoff abbildet, sollten unsere Testpersonen möglichst nahe an dieser Vorlesung sein. Da die Vorlesung nicht im FS21 angeboten wurde, haben wir uns entschieden Studierende aus dem 2. Semester, die das Modul Bsys2 besuchen, anzufragen, da diese das Modul Bsys1 im vorherigen Semester besucht haben. Der Nachteil daran ist, dass Studierende, die das Modul Bsys1 nicht bestanden haben, oder das Informatikstudium bereits nach dem ersten Semester abgebrochen haben, nicht in dieser Gruppe auftauchen. Unter unseren Testpersonen war deshalb wahrscheinlich niemand, der seine Bsys1 Kenntnisse als “schlecht” einstufte.

Die fünf Personen, die sich gemeldet haben, sind männlich und zwischen 21 und 33 Jahre alt und haben verschiedene Ausbildungen vor dem Informatik Studium absolviert. Drei davon haben die technische und zwei die kaufmännische Berufsmaturität abgelegt. Zwei haben zuvor eine Lehre als Informatiker gemacht. Alle Teilnehmer haben den Test einzeln jeweils in einer Konferenz via MS Teams an ihrem eigenen Laptop absolviert und dabei ihren Bildschirm mit uns geteilt. Die meisten Personen hatten dabei ihre Kamera eingeschaltet, so dass wir die Mimik mitverfolgen konnten. Die Testpersonen wurden für ihren Aufwand nicht entlohnt.

Testablauf

Vergleich unseres Simulators mit anderen

Um herauszufinden, ob unser Ergebnis besser für den Unterricht geeignet ist, als verfügbare Produkte haben wir uns überlegt, wie wir die Produkte vergleichen können.
Die erste Möglichkeit wäre, verschiedene Screenshots von Simulatoren zu zeigen, wobei einer davon unserer ist und die Studierenden wählen zu lassen, welchen sie verwenden möchten und welchen sie am besten an die Vorlesung erinnert. Die zweite Möglichkeit wäre,

die Testpersonen zuerst auf einem anderen Simulator ein Programm ausführen zu lassen und dann dasselbe bei unserem zu wiederholen und den Wissensstand zu vergleichen.

Der Nachteil der ersten Methode mit Screenshots ist, dass uns die Erfahrung aus dem Test der Studienarbeit gezeigt hat, dass die Studierenden sich unseren Simulator zu Beginn jeweils ganz genau anschauen, um zu sehen welche Komponenten vorhanden sind und was sie alles erkennen. Wenn wir nun Bilder vor dem Test zeigen, haben sie also unseren Simulator bereits gesehen und wir können keine Aussagen mehr machen darüber, ob sie sich schnell zurecht gefunden hätten im richtigen Test. Führt man diesen Test nach dem Usability Test durch, wissen sie bereits, welcher unser Simulator ist und würden wahrscheinlich eher diesen auswählen. Dieser Test müsste also mit einer zusätzlichen Gruppe durchgeführt werden. Da es sich als schwierig herausgestellt hat, Testpersonen zu finden, haben wir diese Idee verworfen.

Die zweite Methode mit der Ausführung eines Programms in beiden Simulatoren hat zum Vorteil, dass wir sehen können, ob unser Simulator Probleme löst, die der andere nicht lösen kann. Die Schwierigkeit war, einen Simulator zu finden, mit dem man einen fairen Vergleich machen kann. Die Darstellung der Komponenten sollte auf derselben Abstraktionsebene wie in der Vorlesung sein. Ausserdem soll die Syntax möglichst ähnlich wie in der Vorlesung sein. So fielen beispielsweise Simulatoren, die auf dem Little-Man Computer basieren, weg. Unser Simulator hat eine bessere Bedienbarkeit und ein schöneres Design als die anderen, die wir gefunden haben. Bei uns ist durch die Animation klar, was sich warum ändert. Die wenigsten der anderen Simulatoren bieten Animationen an. Des Weiteren wollten wir verhindern, dass die Studierenden merken, dass sie ein fremdes Produkt verwenden. Grund dafür war, dass wir uns vorstellen könnten, dass sie sonst dem fremden Produkt sofort eine schlechtere Bewertung als unserem geben würden.

Davis - Assembly debugger (x86)

Wir haben schlussendlich den “Assembly debugger (x86)” des Projekt Davis[29] von Jakub Beránek für geeignet befunden. Dieser hat eine ähnliche Syntax wie unser, und es ist nicht auf den ersten Blick ersichtlich, dass der Simulator nicht von uns stammt. Er zeigt ein Memory, Register (keine Longsize, sondern Shortsize wie EAX), und Flags an. Die Daten sind jedoch als dezimal statt hexadezimal Werte angezeigt. Im Memory werden keine Adressen

[29] Assembly debugger (x86): https://kobzol.github.io/davis/

angezeigt, dafür bietet es die Funktionalität, dass man die Werte in deren ASCII Interpretation umwandeln kann. Diesen Gedanken hatten wir in der Studienarbeit auch, haben ihn jedoch verworfen, da die Werte meistens keinen Sinn ergeben. Die Instruktionen werden nicht als Maschinencode im Memory dargestellt, man sieht also nicht, was mit dem eingegebenen Code passiert. Dafür wird der Instruction Pointer bei ihm direkt im Register EIP dargestellt und ist somit näher an der Realität als unsere Darstellung. Er bietet im Gegensatz zu unserem Konzept eine Konsole an, wo man zum Beispiel ein “Hello World” ausgeben kann. Ausserdem stehen viel mehr Steuerungselemente zur Verfügung. Sie sind jedoch unserer Meinung nach auf den ersten Blick nicht verständlich, vor allem wenn man nicht alles am Stück abspielen will sondern Schritt für Schritt vorwärts gehen möchte.

Aufgabenstellung im Davis und in unserem Simulator

Wir haben den Studierenden beide Male den Auftrag gegeben, den Code

```
MOV EBX, [0x20]
INC EAX
ADD EAX, EBX
```

auszuführen und herauszufinden, was der genau macht. Nach 4 Minuten haben wir sie dann gefragt, was genau bei der Add-Instruktion passiert und was für Komponenten darin involviert sind. Ausserdem wollten wir von ihnen wissen, woher die CPU die Instruktionen holt, die man im Programm eingegeben hat. Ausserdem sollten sie uns sagen, was ihnen am Design gefällt und was nicht. Dasselbe haben sie dann mit unserem Simulator auch gemacht.

Benutzbarkeit und Verständlichkeit unseres Simulators

Um die Benutzbarkeit zu testen, beobachteten wir die Studierenden in den ersten 4 Minuten, die sie mit unserem Simulator verbrachten. Wir wollten sehen, ob sie das Programm problemlos in den Editor eingeben können und den Simulator starten. Nach dem ersten Test haben wir sie dann gebeten, das Programm zu ändern. Wir haben bewusst die Wörter “zurück” und “Editor” nicht erwähnt, um herauszufinden, ob sie es via Button zurück in den Editor schaffen.

Danach haben wir sie frei mit dem Simulator arbeiten lassen. So konnten wir sehen, ob sie sich zurechtfinden und auch wie sie den Simulator verwenden. Zum Schluss haben wir sie gefragt, ob sie das Stack Programm ausführen können, sofern sie das nicht schon gemacht haben. Wir wollten sie zum Stack befragen und deshalb sicherstellen, dass sie diesen auch

bereits gesehen haben im Programm. Ausserdem haben wir sie gebeten, den Verlauf und den Assembly Code anzuzeigen und Animationen auszustellen, falls sie das nicht schon von allein gemacht haben. Wir wollten so sehen, ob sie die Features gesehen haben und ob sie diese verstehen.

Um zu sehen, ob sie etwas lernen können bei unserem Simulator haben wir ihnen am Ende Theoriefragen gestellt. Zum Beispiel sollten sie den Prozessor-Zyklus beschreiben und wie der Stack funktioniert. Beim Usability Test in der Studienarbeit haben wir bereits im Voraus dieselben Fragen gestellt und die Antworten vor und nach dem Test verglichen. Wir haben uns in diesem Test bewusst dagegen entschieden, um den Studierenden keinen Hinweis zu geben auf was sie besonders achten müssen. Wir wollten ja zum Beispiel sehen, ob sie den Stack von allein nun besser erkennen. Wir haben sie deshalb zu Beginn des Tests nur gefragt, wie der Prozessor-Zyklus abläuft, um zusehen an welche Schritte sie sich erinnern.

Ergebnis Usability Test 2

Davis Simulator

Keiner der Studierenden hat bemerkt, dass es sich beim Davis Simulator nicht um unser Endprodukt handelt. Einige haben nach der Auflösung am Ende des Tests gesagt sie hätten noch viel mehr schlechtes Feedback gegeben, wenn sie gewusst hätten, dass er nicht von uns ist. Es hat sich während des Tests gezeigt, dass sich die Studierenden zwar gewisse Features wie Breakpoints, Animationstempo einstellen, Code direkt sehen oder eine Konsole wünschen, aber das Verständnis für den Stoff und die Bedienbarkeit bei uns viel höher ist.

Beispielsweise sind Änderungen im Davis Simulator nicht animiert, das bedeutet, man hat keine Chance zu sehen, dass etwas passiert ist, wenn sich der Wert nicht ändert. Die Studierenden haben uns als Feedback gegeben, dass die Anzeigen Sinn machen würden, wenn man bereits weiss, was die Instruktionen machen. Die Bedienbarkeit war auch nicht gegeben, da wir oft gefragt wurden, wie man nur einen Schritt vorwärts gehen kann. Wir wussten vor dem Test nicht, dass man Breakpoints setzen kann, das haben wir erst während des Tests erfahren, als ein Teilnehmer diese gesetzt hat.

Bedienung

Da wir den Simulator in einem HTML File ausliefern, und er im Google Chrome Browser ausführbar ist, war das Starten des Simulators kein Problem. Die Teilnehmer haben das File, ohne zu zögern, im Teams Chat heruntergeladen und innert Sekunden geöffnet. Ein Teilnehmer hatte einen Mac und einer Arch Linux, die restlichen Windows, trotzdem brauchte es keine Anleitung, um den Simulator zu starten. Ausserdem wurde der Simulator auf den verschieden grossen Bildschirmen immer schön angezeigt durch das neue Responsive Design, das wir hinzugefügt hatten.

Buttons

Die Bedienung des Simulators ist nach fünf Minuten klar. Die Tooltips wurden von ein paar zwar noch gelesen, es hat aber bei den Aufgaben nie jemand auf einen falschen Button geklickt. Beim Usability Test in der Studienarbeit hatten wir das Problem, dass sich die Studierenden nicht sicher waren für was das Icon im Next Step Button steht. Da wir den Button nun mit einem Text “Next Step” ergänzt haben, war dies kein Problem mehr. Beim Weg zurück in den Editor hat eine Person festgestellt, dass die Menü Buttons prominenter dargestellt sind als der “Back to Editor” Button. In einer Folgearbeit müsste man sich die Darstellung der Menü Buttons noch einmal überlegen. Bei den Toggle Buttons zum Ein- und Ausschalten der Animationen hat jemand bemerkt, dass die Tooltips verwirrend sind. Es wurde jeweils die Aktion beschrieben, die man ausführt, wenn man den Toggle Button umstellt, beispielsweise “Skip animations for this step”, wenn der Toggle Button eingeschaltet war. Das ist verwirrend, weil man nicht weiss, ob es den jetzigen Zustand oder die Aktion beschreibt. Wir haben deshalb die Tooltips noch geändert auf “Animations for this step are enabeld”.

Abbildung E 1 Darstellung der Buttons

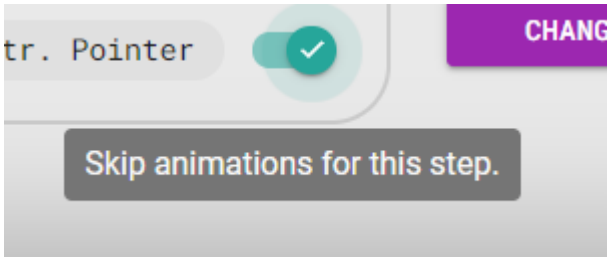


Abbildung E 2 Darstellung des Tooltips beim Skip Animation Toggle Button

CPU Cycle

Die CPU Cycle Box ähnelt durch das Design ohne Schatten und ohne die Hintergrundfarbe nicht mehr den anderen Komponenten, wie beim Usability Test in der Studienarbeit bemängelt wurde. Nun ist jedoch bei den Schritten das Problem, dass man nicht unterscheiden kann, ob der aktuell hervorgehobene Schritt noch ausgeführt wird, oder ob die Animation bereits fertig ist. Man müsste beim aktuellen Schritt noch ein Zeichen oder eine Animation hinzufügen, die nicht ablenkt, so dass klar signalisiert wird, ob er läuft oder abgeschlossen ist.

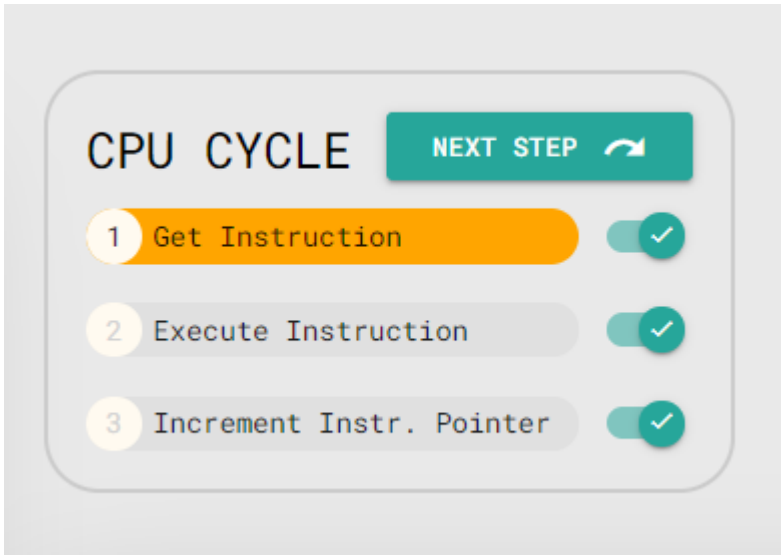


Abbildung E 3 Darstellung des CPU Cycle

Loading Spinner

Ein weiterer Punkt war, dass jemand den Loading Spinner auf dem “Start Program” Button nicht gesehen hat. Er wünschte sich einen Loading Spinner, der über die ganze Seite geht. Wir hatten uns während der Entwicklung aus Zeitgründen dagegen entschieden. Falls dies in einer Folgearbeit implementiert werden sollte, müsste man sich auch überlegen den Zurück-Button im Browser abzufangen, um dort ebenfalls den Spinner über die ganze Seite anzuzeigen. Im Moment erhält man lange kein Feedback, bis dann der Editor geladen wird.

Der Zurück-Button im Browser wurde jedoch von keiner Person verwendet, auch nicht im ersten Usability Test, als der “Back to Editor” Button noch nicht implementiert war.

Abbildung E 4 Darstellung des Loading Spinner

Change Log

Noch nicht optimal war der Change Log. Die Personen haben ihn alle problemlos gefunden und je nachdem auch spezifisch gelobt, wie zum Beispiel, dass die Flags dargestellt werden. Als wir sie später nach der Erwartung gefragt haben, was nach der Ausführung einer Instruktion im Change Log steht, haben nur zwei gesagt die Änderungen. Die anderen erwarten auch die Gelesenen Elemente oder die Vorher/Nachher Werte der Operanden. Jemand erwartete zusätzlich noch den Wert des Instruction Pointers. Wir haben die Idee, alle möglichen Werte zu zeigen, in der Design Phase verworfen, da wir davon ausgegangen sind, dass so zu viele Informationen auf einmal dargestellt werden. In einer Folgearbeit müsste man sich also überlegen, wie man den Change Log besser darstellen könnte und auch testen, ob das verständlich ist.

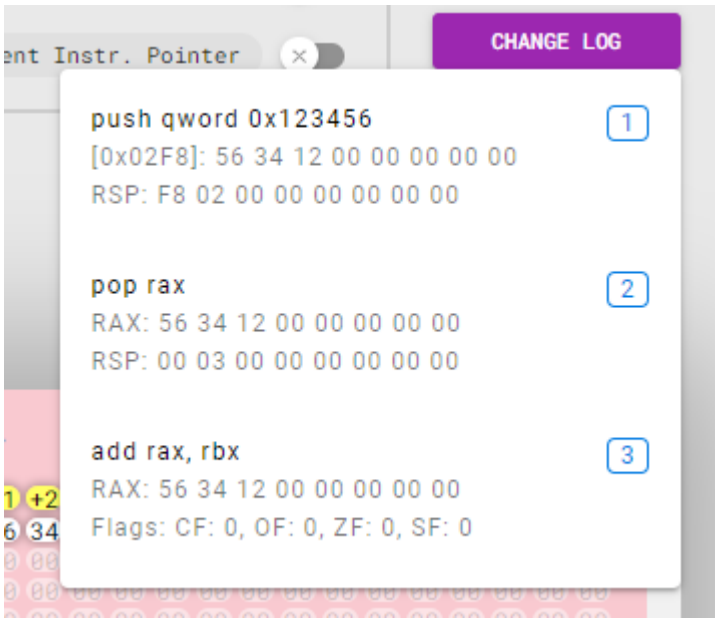


Abbildung E 5 Inhalt des Change Log

Error Handling Simulator

Wir haben noch kein Error Handling implementiert. Wenn ein Fehler auftritt, dass beispielsweise eine Instruktion im Unicorn nicht ausgeführt werden kann, weil man auf unmapped Memory zugreift, wird dieser in der Browser Konsole ausgegeben und der Simulator bricht ab und zeigt den Restart Button. Obwohl sie den Fehler in der Konsole nicht gesehen haben, haben alle sofort realisiert, dass etwas mit ihrem Programm schiefgelaufen ist, als der Restart Button angezeigt wurde. Trotzdem sollte man die Fehlermeldungen sauber in einem Dialog anzeigen.

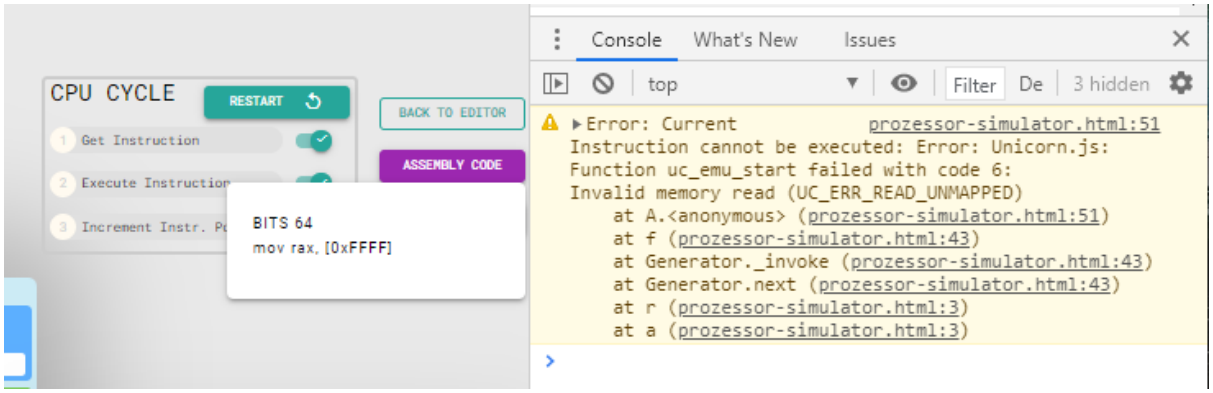


Abbildung E 6 Error Handling im Simulator

Error Handling Code Editor

Einige der Studierende haben eigene Programme geschrieben, oder aktiv versucht fehlerhaften Code auszuführen und haben dabei unsere Fehlermeldung im Code Editor entdeckt. Unabhängig davon, ob sie die Fehlermeldung absichtlich oder unabsichtlich provoziert haben, haben alle die Fehlermeldung sofort gesehen. Auch als sie versucht haben den Fehler zu korrigieren und die Fehlermeldung erneut erschien, haben sie die Meldung wahrgenommen.

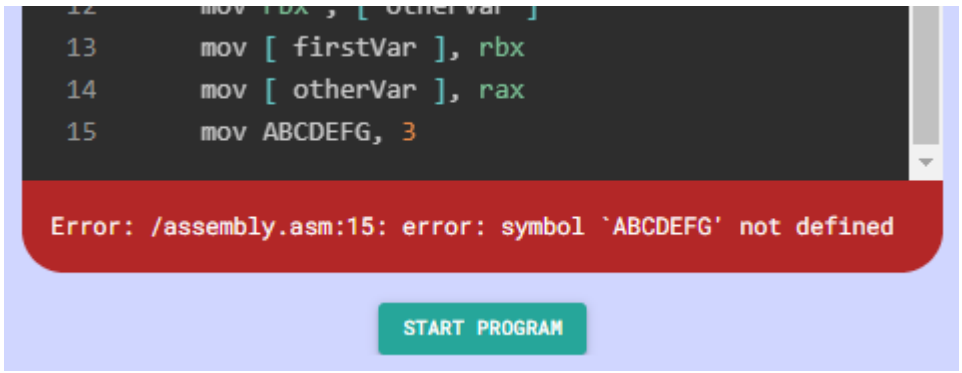


Abbildung E 7 Fehlermeldung im Code Editor

Verständnis des Stoffs

Der Prozessorzyklus wurde schnell erkannt und von den Studierenden als gut dargestellt befunden. Die Funktion der CPU-Box wurde ebenfalls verstanden. Ausserdem hat sich jemand gewünscht, dass wenn man einen Operanden, beispielsweise RAX einliest und denselben wieder als Output hat, dass man dann die Bytes, die geändert wurden, markiert im Output. Die Darstellung der Immediate muss noch einmal überdacht werden. Diese wurde von zwei Personen für einen Fehler gehalten. Eine einfache Lösung könnte vielleicht auch ein Tooltip sein.

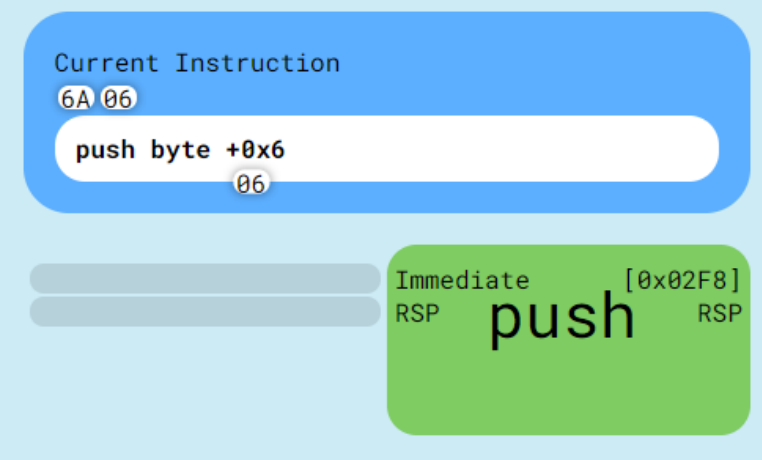


Abbildung E 8 Darstellung der Immediate

Stack Pointer Register

Bei der CPU-Box könnte man sich noch überlegen einen Tooltip für RSP anzubieten, da man nicht weiss, dass es sich um ein Register handelt und deshalb mit der Maus nicht über das Register RSP fährt und so herausfindet, dass es sich um den Stackpointer handelt. Ausserdem hatte eine Testperson den Wunsch, dass wenn man mit der Maus über das Register RSP fährt, das Memory zum Stackpointer Byte scrollt und dort ebenfalls die Tooltips einblendet.

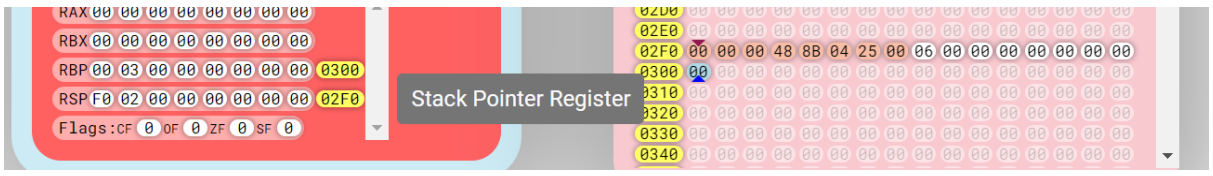


Abbildung E 9 Darstellung des Stack Pointer Tooltip

Darstellung Stack

Der Stack wurde von allen verstanden. Drei Personen haben erwähnt, dass man hier schön sieht, dass die Elemente nicht gelöscht werden, wenn man Pop macht, sondern nur der

Pointer verschoben wird. Wir hatten das nicht bewusst so überlegt, sondern verdanken diesen Effekt unserer Darstellung der Bytes, die bereits verwendet wurden. Was bei unserem Design nicht klar war, ist dass der Stack Pointer immer um 8 Byte erhöht oder verringert wird.

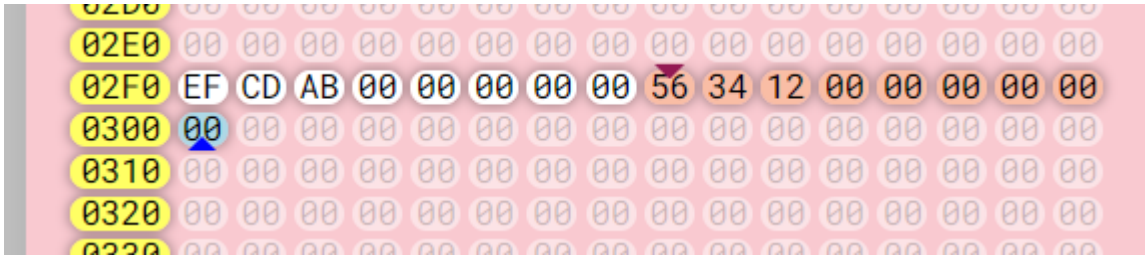


Abbildung E 10 Darstellung des Stacks im Memory

Unterregister

Einer Person war unklar, dass es sich beim Register EBX um gewisse Bytes des Registers RBX handelt. Deshalb wurde EBX auch nicht unten in den Registern angezeigt. Wir haben uns in dieser Arbeit bereits Gedanken darüber gemacht, wie man die Unterregister im Simulator darstellen könnte. Wir könnten uns vorstellen, dass man beim Hovern über die Register eine grosse Animation sieht, die alle Unterregister anzeigt, oder ein Pop-up, wenn man auf das Register klickt. Da man bei der CPU-Box jedoch nicht weiss, dass das Register EBX zu RBX gehört, kommt man aber vielleicht nicht auf die Idee mit der Maus über das Register zu fahren. Deshalb könnte man sich überlegen auch dort eine Tooltip in der CPU-Box anzubieten, dass das anzeigt zu welchen Bytes im RBX das EBX gehört. Eine weitere Idee, die wir hatten, war, neben dem Editor eine Tabelle mit den Registern anzuzeigen, da es beim Programmieren auch nützlich wäre zu sehen, was es für Register gibt.

Erhöhung Instruction Pointer

Die meisten haben gewusst, dass der Instruction Pointer um die Länge der Instruktion erhöht wurde, die wenigsten waren sich jedoch sicher dabei. Bei unserem Design Ansatz in der Studienarbeit haben wir die Erhöhung mit einem Zähler dargestellt, der hochzählt, wenn die Bytes in ihm verschwinden. Danach wird er zur Instruction Pointer Adresse hinzugezählt. Diese Animation haben alle verstanden. Wir haben sie aus Zeitgründen in der Bachelorarbeit vereinfacht, in dem die Bytes einfach zusammen zum Pointer fliegen und er dann aktualisiert wird. In einer Folgearbeit müsste man den Zähler noch hinzufügen und die Bytes einzeln nach unten fliegen lassen.

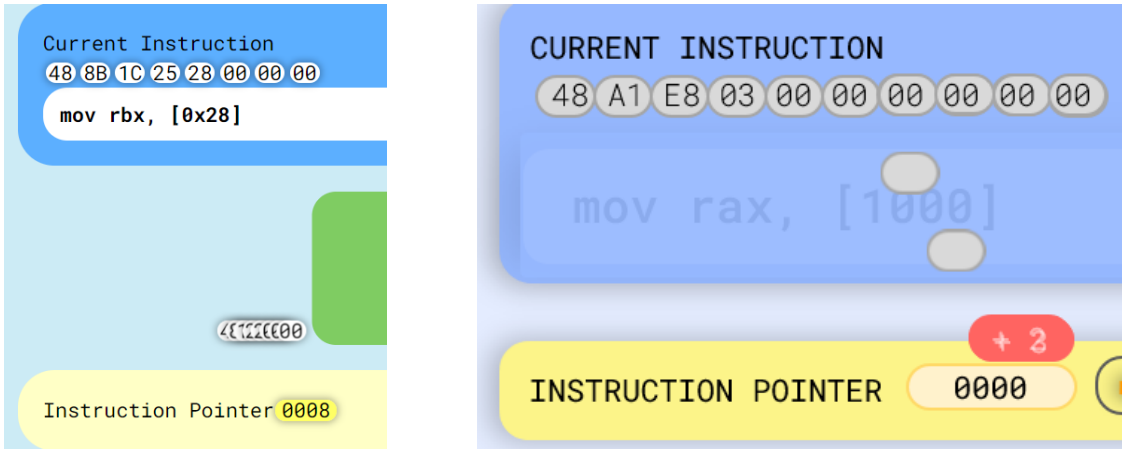


Abbildung E 11Gegenüberstellung der Animation "Increment Instruction Pointer" jetzt (links) und im Designansatz der Studienarbeit (rechts)

Jump und Call Instruktion

Von einigen Studierenden wurde die Jump Instruktion ausgeführt. Diese ist jedoch nicht fertig implementiert. Im Moment wird je nach Jump Instruktion das Zero-Flag gelesen und die Immediate mit der Zieladresse. Danach fährt die CPU-Box nach links und da kein Output erscheint direkt wieder nach rechts. Der Instruction Pointer wird dabei nicht angepasst. Danach kommt im nächsten Schritt die normale Increment Instruction Pointer Animation, die den Eindruck gibt, dass er um die Anzahl Bytes der Instruktion hochgezählt wird. Er wird dann aber auf den Wert der Zieladresse gesetzt. Die einzige Person, die sich nicht beschwert hat, hatte die Animation des Increment Instruction Pointer Schritt deaktiviert.

Um das umzusetzen, müsste man eine eigene Increment Instruction Pointer Animation entwickeln, die ausgeführt wird, wenn ein Jump oder ein Call ausgeführt wird. Vielleicht könnte man die Überschreibung des Instruction Pointer Registers RIP durch die Jump- und Call-Instruktionen im Capstone abfangen. Ansonsten könnte man die Opcodes der verschiedenen Jump- und Call-Funktionen bei den Informationen mit den Operanden Informationen des Instruction Objekts vergleichen.

Verbesserungsvorschläge

Assembly Komponente und Breakpoints

Einige Studierende haben sich eine bessere Übersicht über den Code gewünscht. Man kann den Assembly Code zwar öffnen, er ist aber nicht konstant ersichtlich, ausserdem sieht man nicht, wo man sich aktuell befindet und es gibt kein Syntax Highlighting. Ausserdem ist es

ungeschickt, dass sich der Code über dem Memory öffnet, falls man zum Beispiel beim Swap Programm die Variablen im Code suchen möchte. Des Weiteren haben sich Studierende gewünscht, dass man bei grösseren Programmen nicht schrittweise durchgehen muss, sondern Breakpoints setzen kann.

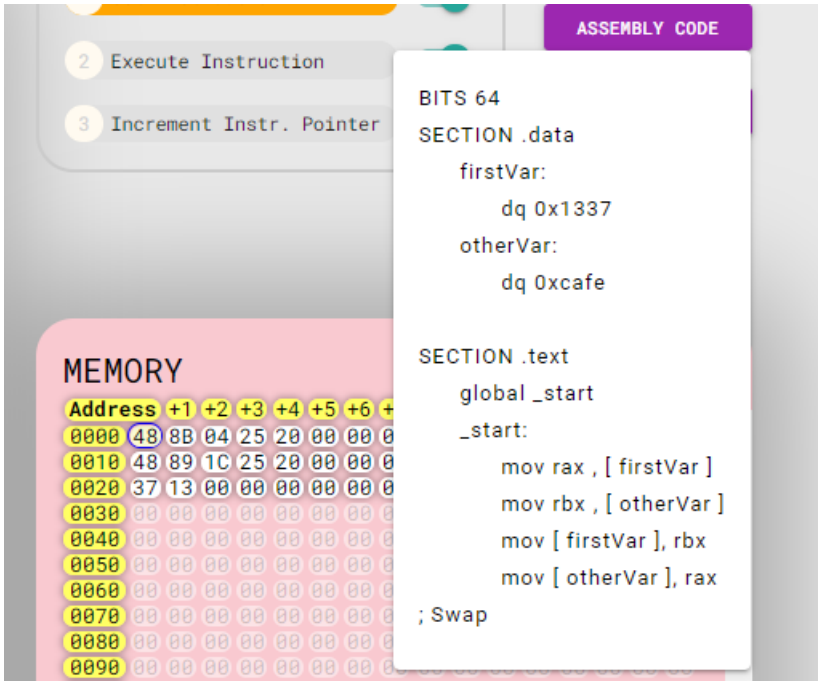


Abbildung E 12 Darstellung des Assembly Codes

In unserem Konzept haben wir nur die Maschinen Bytes im Memory angezeigt, damit klar ist, dass die CPU nicht mit Assembly Code arbeitet. Eine Assembly Übersetzung, die konstant eingeblendet ist, müsste sich also klar von den anderen Komponenten unterscheiden. Ausserdem müsste sie sich jeweils anpassen, sobald es Änderungen im Code Bereich gibt, oder an den Codebereich etwas angehängt wird im Memory. Man kann dies beim Memory Write Access Hook überprüfen und gegebenenfalls aktualisieren. Eine Darstellungsmöglichkeit wäre, dass man noch einmal die Code Editor Komponente darstellt und diese für die Bearbeitung sperrt. So hätte man auch dasselbe Syntax Highlighting wie beim Editor.

Breakpoints müssten ebenfalls im Simulator bei einer Assembly Komponente gesetzt werden. Im Editor ist dies nicht möglich, da der Nasm Optimierungen vornimmt, wenn der den Assembly Code in Maschinenbytes umwandelt. Deshalb ist ein 1:1 Mapping von eingegebenen Assembly Instruktionen zu Maschinencode im Memory nicht möglich. Man müsste also

den erhaltenen Maschinen Code zurück nach Assembly disassemblieren. Mit diesem Assembly Code wäre ein 1:1 Mapping zum Memory Inhalt möglich. Dann könnte man dort Breakpoints setzen und die Ausführung aller Schritte am Stück stoppen, sobald der Instruction Pointer mit der Adresse eines Breakpoints übereinstimmt.

Des Weiteren bräuchte es eine Möglichkeit, dass man selbst Adressen angeben kann, an denen man Breakpoints setzen möchte. Man kann im Simulator Code an eine Stelle im Memory kopieren und dann mithilfe von Call an diese Stelle springen und die Instruktion ausführen. Diese wäre also dann nicht von Anfang an in der Assembly Komponente ersichtlich, man möchte dort aber vielleicht trotzdem Breakpoints setzen.

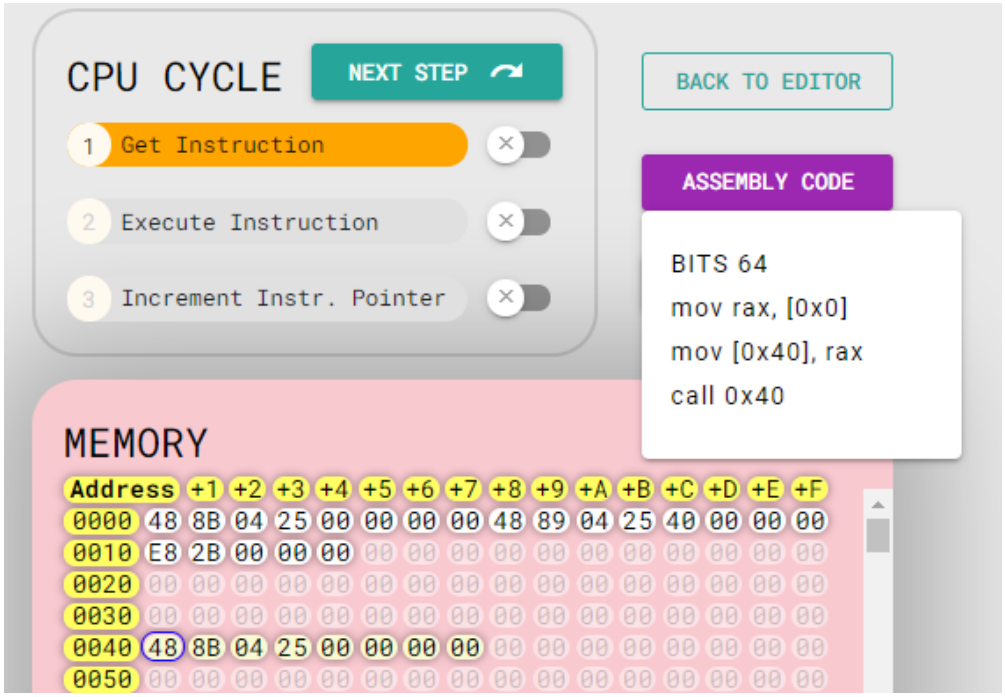


Abbildung E 13 Sprung aus dem Codebereich

Geschwindigkeit einstellen

Die Studierenden haben beim Davis Simulator entdeckt, dass man die Tickrate einstellen kann und möchten das auch bei unserem Simulator. Einige möchten die Animationen beschleunigen, andere möchten sie verlangsamen. Sie möchten ebenfalls einen Slider wie beim Davis Simulator. Der Usability Test hat jedoch gezeigt, dass der Slider mit der Tickrate nicht optimal ist. Erstens ist die langsamste Stufe viel zu schnell und zweitens erwarteten die Studierenden, dass sich die Geschwindigkeit nach links verlangsamt. Da es sich um die Tickrate handelte, funktioniert der Slider in die andere Richtung. Falls man so einen Slider hinzufügen möchte, sollte man also die Geschwindigkeit angeben und nicht die Tickrate.

Syscalls und Interrupts

Ein Student hat versucht ein Hello World Programm aus dem Internet auszuführen. Der Interrupt hat einen Fehler im Unicorn verursacht, da Syscalls und Interrupts in Unicorn nicht implementiert sind. Durch die Fehlermeldung bricht der Simulator ab und der Restart Button erscheint. Unicorn bietet einen Hook für Interrupts und Syscalls, den man hinzufügen kann. Würde man einen leeren Hook einbauen, würde der Simulator nicht mehr gestoppt werden, da von Unicorn kein Fehler mehr geworfen würde. In der Callback Funktion von diesem Hook könnte man einen Dialog anzeigen, der informiert, dass Syscalls und Interrupts nicht möglich sind. Man könnte aber auch die Interrupt und Syscall Funktionalität implementieren und beispielsweise eine Konsole anzuzeigen.

Abbildung E 14 Hello World Programm

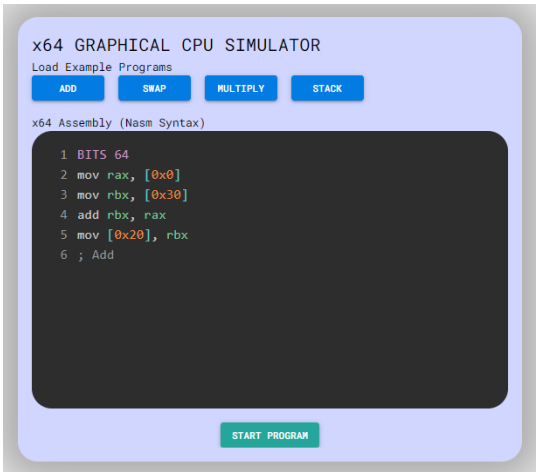
Onboarding und Hilfe

Die Studierenden brauchten einen kurzen Moment, um sich im Simulator zurechtzufinden. Ausserdem gibt es wie bereits erwähnt, gewisse Schwierigkeiten, die Unterregister zu erkennen oder bei der ersten Durchführung einer Push Instruktion RSP als Stackpointer Register zu erkennen. Um dem entgegenzuwirken könnte man ein Onboarding einführen, dass die wichtigsten Komponenten nacheinander einblendet und kurz beschreibt. Des Weiteren könnte man ein Hilfe Button machen, der alle Tooltips einblendet. So erkennt man vielleicht schneller, dass man über das RSP Register hovern kann.

Anhang F Anleitung Prozessor-Simulator

Um den Prozessor-Simulator zu starten, braucht man das HTML-File. Dieses kann zum Beispiel über den Skripteserver oder per E-Mail ausgeliefert werden. Nach dem Herunterladen kann das HTML-File im Google Chrome Browser gestartet werden. Leider wurde der Prozessor-Simulator nur für diesen Browser entwickelt.

Im Code Editor kann Assembly Code aus der Vorlesung eingegeben werden. Am besten startet man jedoch mit einem der angebotenen Beispielprogrammen.



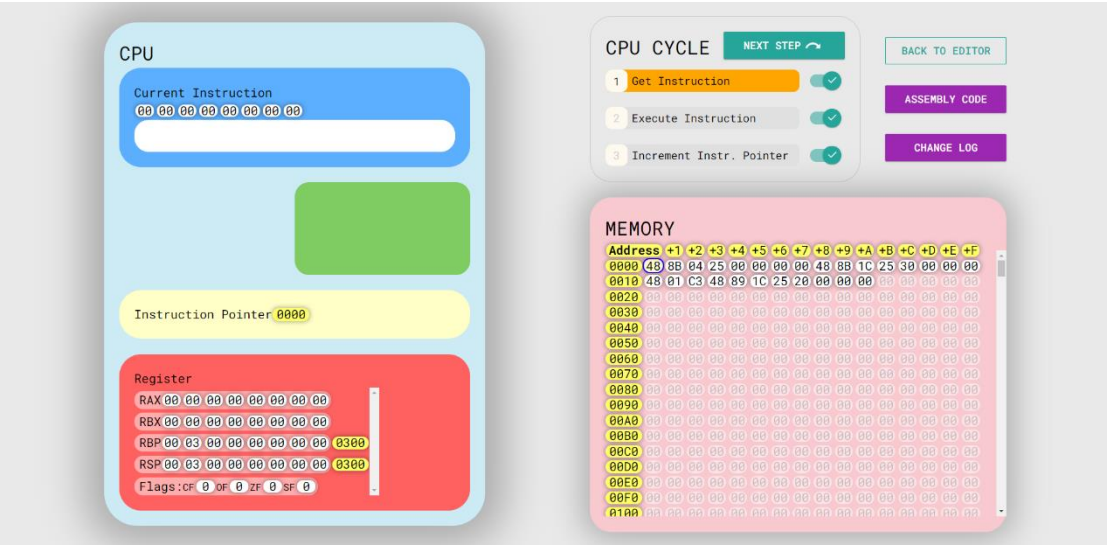
Man muss beachten, dass zu Beginn des Programms jeweils der Modus “BITS 64” angegeben werden muss. Ausserdem werden Syscalls und Interrupts nicht unterstützt.

Mit dem Button “Start Program” wird der Simulator gestartet, sofern syntaktisch korrekter Code eingegeben wurde.

Achtung: Es lohnt sich den eingegebenen Code zum Beispiel in einem Text File auf dem Computer abzuspeichern, da er von der Website nicht gespeichert wird.

Im Simulator kann man sein Programm mit dem “Next Step” Button schrittweise ausführen. Falls man einen Schritt bereits verstanden hat, kann man die Animation mit dem Toggle Button neben dem Schritt umstellen. Wenn man den Code ändern will, gelangt man mit dem “Back to Editor” Button wieder zum Code Editor zurück.

Achtung: Leider wurde kein Fehlerdialog implementiert, deshalb werden Fehlermeldungen in der Konsole des Web Inspectors ausgegeben.



Anhang G Ideen für Erweiterungen

Während der Entwicklung haben wir Ideen für Erweiterungsmöglichkeiten gesammelt, und uns überlegt, wie man diese umsetzen könnte. Dasselbe machten wir auch für die Verbesserungsvorschläge des zweiten Usability Tests. Die entsprechenden Umsetzungsideen sind im Anhang E beschrieben.

Spezifische Instruktionen animieren

Um neben der generischen Animation der Instruktionen auch noch eine spezifische für jede einzelne anzubieten, könnte man ein Pop-Up hinzufügen. Unsere Idee wäre, dass wenn die CPU-Box sich bewegt und die Instruktion verarbeitet, dieses Pop-Up geöffnet wird und man sehen kann, was die CPU mit der Instruktion macht. Zum Beispiel könnte man bei einem „add“ die Addierung der einzelnen Bits animieren. Anhand des Opcodes, den man im OperandsInformation Objekt der Instruktion erhält, könnte man erkennen, um welche Funktion es sich handelt. Dann könnte man im AnimationController, wenn die CPU-Box sich bewegt, die richtige Animation starten.

Speicherbus hinzufügen

In der Aufgabenstellung und auch in den Usability-Tests wurde gewünscht, dass man den Speicherbus sehen kann. Dieser ist auch eine wichtiger Inhalt der Vorlesung. Leider reichte uns die Zeit nicht, um diesen umzusetzen. Es wäre jedoch möglich, ihn in einer Folgearbeit hinzufügen. Dafür müsste man eine neue Komponente im Vue erstellen, die man dann zwischen dem Memory und der CPU darstellen würde. Bei einem Lesezugriff auf das Memory müsste man die animateMemoryRead Funktion so abändern, dass die Adresse der MemoryLine auf den Speicherbus gelegt wird. Diesen müsste man dann so animieren, dass er an dieser Adresse die MemoryLine holt, und an die CPU liefert, die den Wert bei den Inputs ablegt. Diese Animation müsste man auch im Schritt “Get Instruction” hinzufügen. Beim Schreibzugriff würde man dann die Adresse und den Wert aus dem Output auf den Speicherbus legen. Dieser würde nachher den Wert an die richtige Stelle schreiben.

Verschiedene Speicherbereiche anzeigen

Eine Idee aus der Aufgabenstellung war, dass man verschiedene Speicherbereiche anzeigen kann. Man könnte also im Memory gewisse Bereiche beobachten, während dem die Animationen im Memory automatisch scrollen. Man könnte zum Beispiel weitere Memory Komponenten öffnen mit einer Kopie des Memory, wo man an die gewünschte Stelle scrollen kann. Hier müsste man jedoch beachten, dass die Kopien eigene einziger IDs haben. Zum Beispiel werden die LocationIds der Bytes für die Animationen verwendet. Man darf also nicht mehrere Elemente mit derselben identischen ID im HTML anzeigen. Ausserdem möchte man nicht, dass die Animationen in der kopierten Speicherbereich stattfinden oder diese automatisch gescrollt werden. Zusätzlich wäre es möglich, dass man im Editor angeben kann, wo der Speicherbereich starten soll. Dies könnte man dann dem StartSimulator-Service übermitteln.

Operanden Informationen anzeigen

Ein Wunsch aus dem 1. Usability-Test und auch aus der Studienarbeit war, dass man in CurrentInstruction bei den Operanden im Assembly Code oder auch im Maschinencode mehr Informationen erhalten kann. Von Capstone erhalten wir bereits gewisse Informationen, über die Operanden, die man in Tooltips anzeigen könnte. Wir wissen zum Beispiel, wenn der zweite Operand ein Register ist und könnten auch angeben, ob das Register in einem anderen liegt (z.B. EAX in RAX).

Schwieriger wird es, wenn man die Bytes des Maschinencodes interpretieren möchte. Man müsste einen Mechanismus finden, um zu erkennen, welcher Teil des Codes den Opcode für die Instruktion angibt und wie viele und wie grosse Operanden dazugehören. In der Studienarbeit haben wir im Ausblick festgestellt, dass ein wichtiger Schritt im Prozessor-Zyklus fehlt, weil die CPU die Instruktionen nicht auf einmal liest, sondern immer Byte für Byte. Anhand des Opcodes weiss sie zum Beispiel dann, wie gross die Operanden sind. Würde man diese Idee umsetzen, könnte man auch die Tooltips für den Maschinencode anbieten.

Alle Register anzeigen

Im Moment werden die Register im Simulator angezeigt, sobald sie verwendet werden. Vielleicht möchte man jedoch ein spezifisches Register von Anfang ansehen. Im StartSimulator-Service geben wir einen Array mit RegisterIDs an, die von Anfang an im Simulator angezeigt werden sollen. Es wäre also möglich, dass man ihn Editor, eine Liste mit allen verfügbaren Register zur Verfügung stellt, wo der Benutzende die gewünschten Register angeben kann. Diese Liste könnte man dann an den Simulator übergeben, der die dann anzeigen soll. Man müsste jedoch beachten, dass die Benutzenden merken, dass auch automatisch Register hinzugefügt werden und sie diese nicht jedes Mal alle angeben müssen.

Ausführung steuern

Um die Ausführung zu starten, so dass die Animationen eins nach dem anderen durchlaufen und erst bei einem Klick auf einen Stop Button abbrechen könnte man die Signalisierung in der Controls Komponente anpassen. Der Button würde ein Signal an den Controller schicken, welche mit Hilfe von einer privaten Boolean Variable den Modus speichert, dass man nicht anhalten möchte. Am Ende von einem Animationsschritt wird diese Variable überprüft und falls gesetzt, werden direkt die Animationen vom nächsten Schritt gestartet. Für die Animationsgeschwindigkeit könnte man in den Controls einen Slider oder Buttons einbauen, die eine Zahl an den Controller schicken. Der Controller setzt dann eine globale CSS-Variablen für die Geschwindigkeit, die jeweils in den CSS-Animationsdauerregeln verwendet werden würden (siehe Screenshots).

```css
:root {
  --moveByteAnimationDuration: 5s;
}
```

```css
.moveByte {
  animation: animationMoveByte 2s;
}

.moveByte {
  animation: animationMoveByte var(--moveByteAnimationDuration);
}
```

Anhang H Ergebnisse Code Analyse

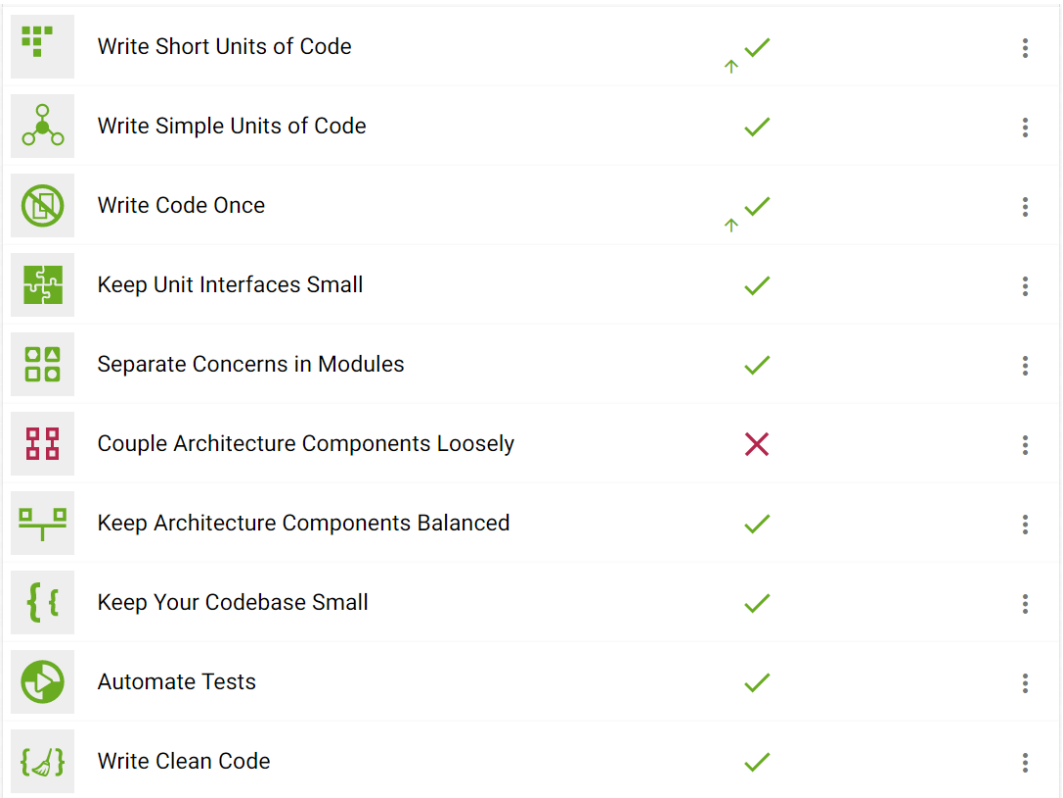
Ergebnisse Codeanalyse

Datum: 03.06.2021

Tester: Eliane Schmidli

Hier werden die Ergebnisse der Codeanalyse Tools dokumentiert.

Ergebnis Better Code Hub

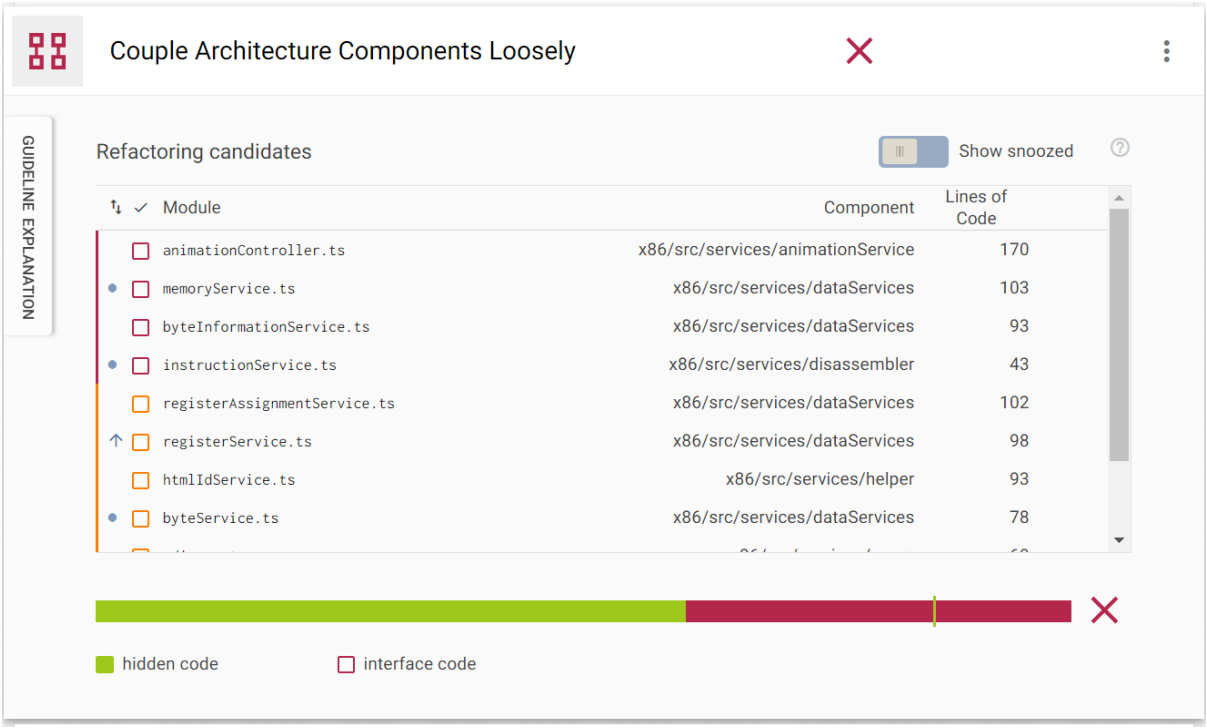


Details aus den Kategorien:

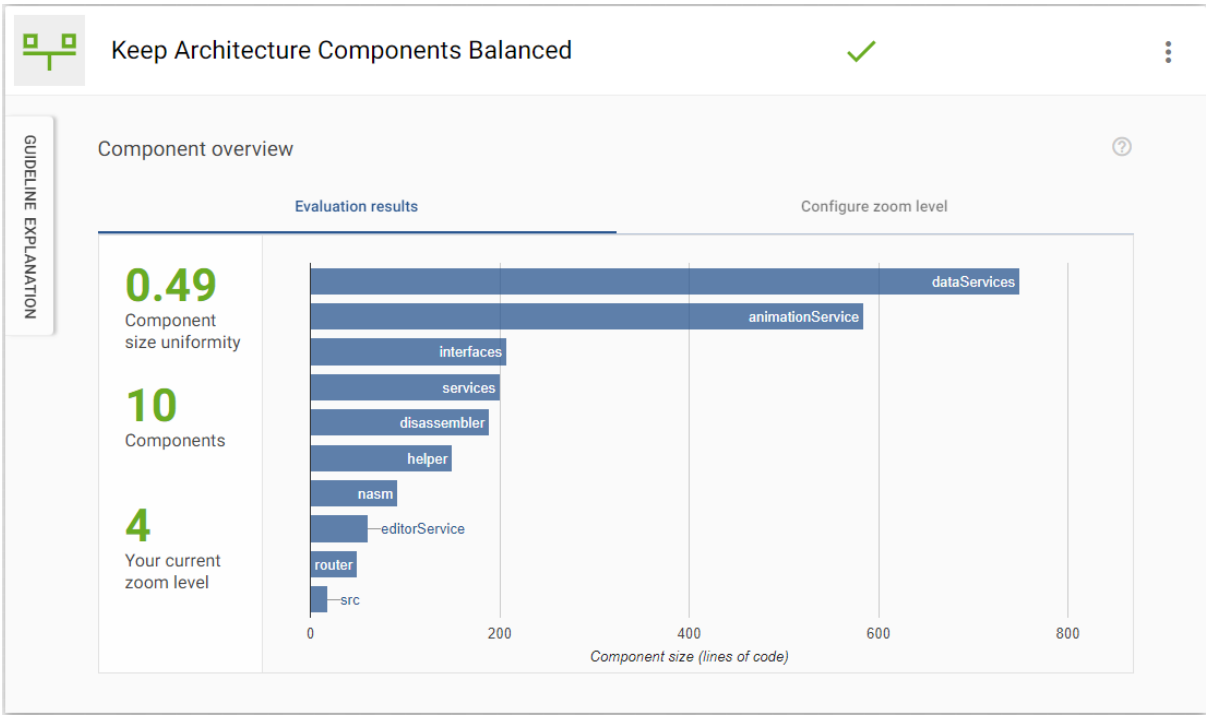
Längste analysierte Methode: StepController.nextStep, 27 Lines of Code (LoC)

Maximale Anzahl Parameter: 3

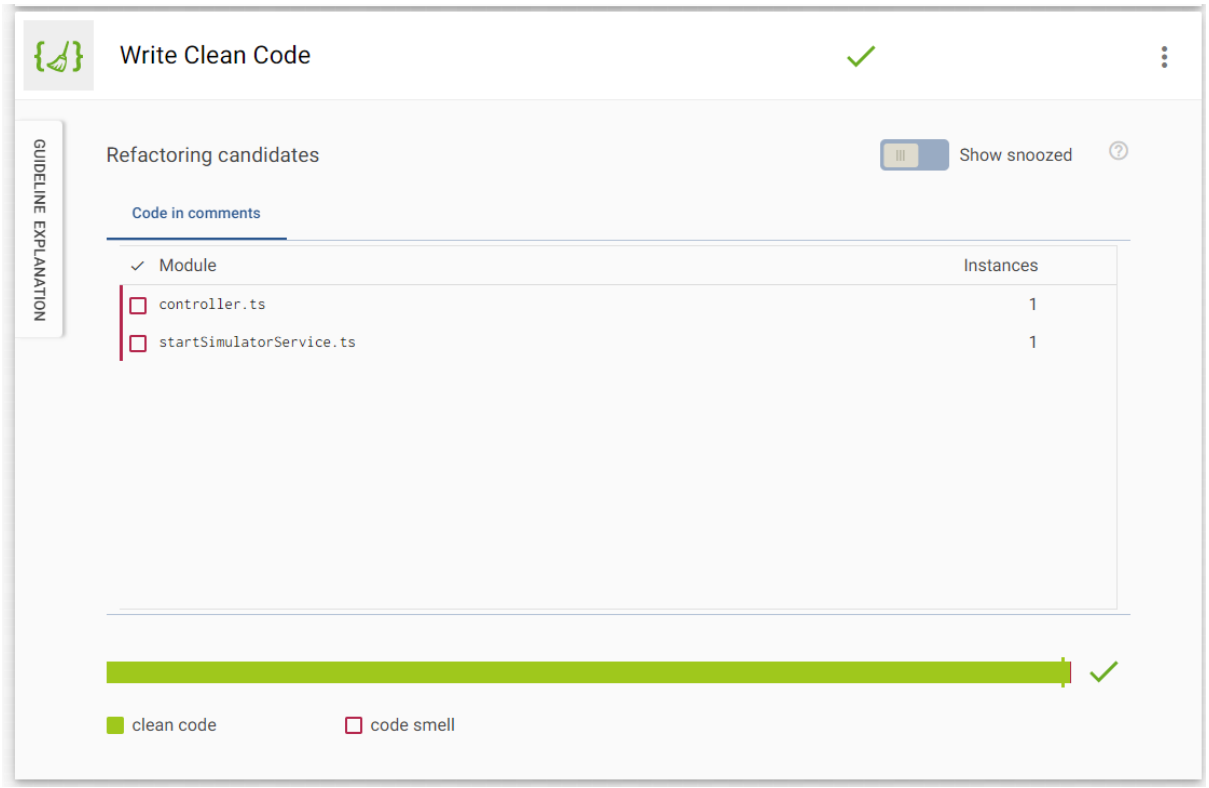
Meiste Branch Points: instructionOperandsService.ts.createPointerArithmeticOperands, 8 Branch Points



“Couple Architecture Components Loosely” haben wir wegen des animationController, memoryService, ByteInformationService und dem instructionService nicht erreicht.



Oben sieht man die Komponenten, die Better Code Hub in die Analyse einbezogen hat.



Bei “Write Clean Code” haben wir zwei Refactoring Kandidaten erhalten. Dabei handelt es sich jedoch um unser provisorisches Error Handling, dass die Fehlermeldungen in der Konsole ausgibt.

```
154     } catch (e) {
155       /* eslint no-console: ["error", { allow: ["warn"] }] */
156       console.warn(e);
157       return false;
158     }
159     break;
```

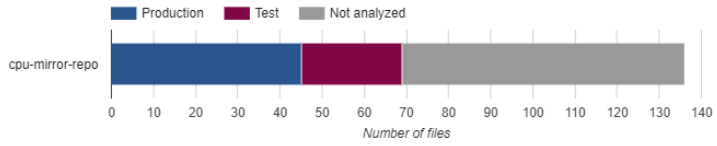
Unten sieht man die Konfiguration, die wir verwendet haben. Wie man sieht, werden die Vue-Komponenten nicht analysiert, da Better Code Hub diese nicht versteht.

Analysis configuration

The previous analysis ran with the following configuration:

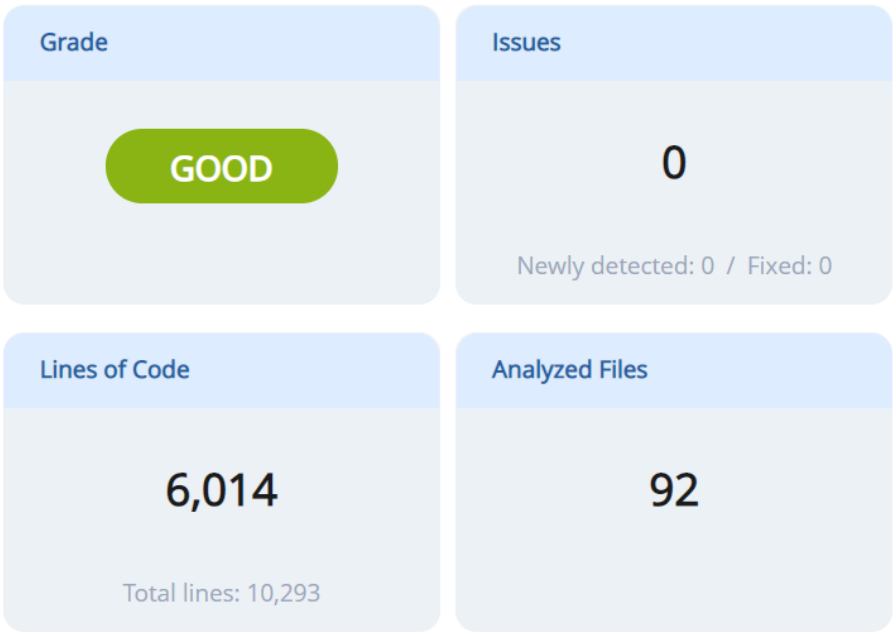
```
component_depth: 4
default_excludes: true
languages:
  - typescript
  - javascript
exclude:
  - .*/emulatorEnums\.ts
  - .*/emulatorService\.ts
  - .*/disassemblerEnum\.ts
  - .*/disassemblerService\.ts
  - .*/nasm\.js
  - .*/ndisasm\.js
  - .*/licensesToShow\.ts
```

The following chart shows how many files were categorized as Production, Test and Not analyzed for the previous analysis.



To further configure your Better Code Hub analysis, follow the instructions here. Rerun the analysis to see the results.

Deepscan



In der Studienarbeit hatten wir 2020 LoC und 55 analysierte Files. Im Vergleich zur Studienarbeit hat sich die Anzahl LoC ungefähr verdreifacht.

Name ▲	Issues	Lines of Code	Total Lines
x86 ✖	0	6,014	10,293
..			
src ✖	0	3,701	7,377
tests/unit/components ✖	0	527	697
tests/unit/services ✖	0	1,786	2,219

Konfiguration:

Exclude files from your analysis.

```
/x86/vue.config.js
/x86/babel.config.js
/x86/.eslintrc.js
/x86/src/router
/x86/src/quasar.js
/x86/src/main.ts
/x86/tests/unit/services/testDataNextState.ts
/x86/tests/unit/services/testDataCurrentState.ts
/x86/src/services/disassembler/disassemblerEnum.ts
/x86/src/services/emulator/emulatorEnums.ts
/x86/src/components/licenseButton/licensesToShow.ts
```

Anhang I Ergebnisse Webseitenanalyse

Ergebnisse Webseiten Analyse

Datum: 14. Juni 2021
Tester: Yves Boillat

Hier werden die Ergebnisse der Webseiten Analyse mit Hilfe von Browser Tools dokumentiert. Die Tests wurden mit dem Endstand des Codes durchgeführt.

Taskmanager

Der Taskmanager zeigt auf dem Zielsystem (Laptop mit CPU: Intel i7 8565U, 16 GB RAM, Bildschirmauflösung: 1920 x 1080, Windows 10) für den Chrome-Prozess des Simulators eine maximale CPU-Nutzung von 18%.

Ergebnis Chrome Performance Monitor

Um die Performance der Webseite während aktiver Nutzung zu messen, wurden 100 Sekunden mit dem Chrome Performance Monitor analysiert. Die analysierte Zeitspanne beinhaltet das Laden vom Simulator, sowie drei ganze Prozessor-Zyklus Animationen. In den ersten 10 Sekunden wird die Seite geladen und befindet sich dann im Ruhezustand und wartet auf eine Eingabe. Beim Zeitabschnitt 12:00:07 wird die erste Animation gestartet. Nach jedem Schritt wird nach circa 3 Sekunden Wartezeit der nächste Schritt ausgeführt.
Man sieht wie der DOM ziemlich konstant um die 40’000 Nodes enthält und nur während den Animation neue Nodes dazu kommen und nach einer kurzen Zeit wieder gelöscht werden. Der JavaScript Heap wächst bei jedem Schritt des Prozessor-Zyklus und nimmt wieder ab.Über eine längere Zeit scheint er nicht grösser zu werden. Man sieht, dass aufgrund der Animationen sehr viele Style Berechnungen durchgeführt werden. Die periodischen Layout Berechnungen, die den Byte-Bewegungs-Animationen zu verdanken sind, sind auch ersichtlich. Auch die CPU-Nutzung scheint mit den Style- und Layout-Berechnungen zu korrelieren. Die 100% Auslastung wird nie überschritten, was auf dem Mac-System, auf dem getestet wird, einem voll ausgelasteten CPU-Kern entspricht. Die durchschnittliche CPU-Auslastung

ist jedoch nicht sehr hoch da die Leistung nur vor und während der Animationen benötigt wird.

Abbildung I 1 Chrome DevTools Performance Monitor

Ergebnis Chrome Performance Tab

Auf der Zeitachse bei 84 ms wir die Webseite von Editor geladen. Bei ~6000 ms wird auf die Simulator-Seite gewechselt, die bei ~9000 ms fertig geladen ist. Danach wird bis zum Ende der Zeitachse in normalem Tempo durch die Prozessor-Zyklus-Schritte geklickt.

Auf dem ersten Graphen sieht man in hellgrün die Frames per Second (FPS) über die Zeitachse. Der Wert geht von 0 bis 60 (oder höher). Im Ruhezustand liegt die FPS bei 0 da nichts aktualisiert wird. Vor einer Animation schwankt der Wert jeweils Zwischen 1 und 65 FPS. Sobald die CSS-Animationen fertig berechnet sind und starten liegt der Wert zwischen 35 und 65 FPS.

Im zweiten Graphen sieht man in violett die CPU-Nutzung. Auf der y-Achse sind keine konkreten Wertangaben vorhanden. Im Ruhezustand liegt die Auslastung bei 0. Beim Seitenaufbau bei ~8500 ms geht die CPU-Nutzung auf den Maximal Wert. Jeweils vor und oft auch

während der CSS-Animationen steigt der Wert auf die maximale Auslastung und fällt dann wieder auf 0. Der Grund für die Auslastung ist das Berechnen vom "Layout Tree".

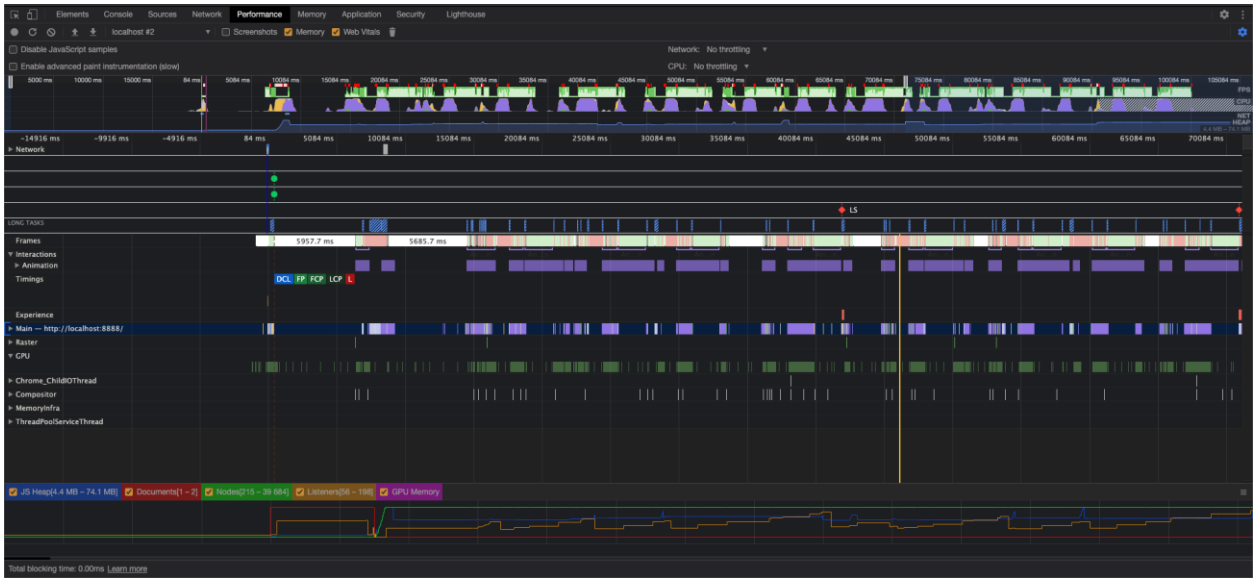


Abbildung I 2 Chrome DevTools Performance Tab

Ergebnis Safari Timelines Test

Mit Safari können wir die durchschnittliche CPU-Nutzung messen. Wir nutzen die Timelines Funktion und messen 63 Sekunden lang die CPU-Nutzung. Bei Sekunde 2 wird der Editor geladen und bei Sekunden 8 wird der Simulator gestartet. Bis Ende der Zeitachse werden in normalem Tempo die Prozessor-Zyklus-Schritte durchgeklickt.

Auf der "Enegry Impact" Skala ist ersichtlich, dass die durchschnittliche CPU-Nutzung bei 38.1% liegt. Da auf einem Mac getestet wird, entspricht dieser Wert der Auslastung eines CPU-Kerns. Da auf dem System zwei Kerne verbaut sind, beläuft sich der Wert auf 19.05% der Systemauslastung. Laut "Main Thread" Anzeige sind 52.6% der Auslastung auf das Layout zurückzuführen.

Abbildung I 3 Safari Timelines Tab

Ergebnis Chrome Frame Rendering Stats

Der Live Frame Rate Monitor in Chrome zeigt während einer Animationen Werte zwischen 35 und 65 FPS an. Vor dem Start der Animation bricht der FPS Wert ein und fällt unter 30 FPS.

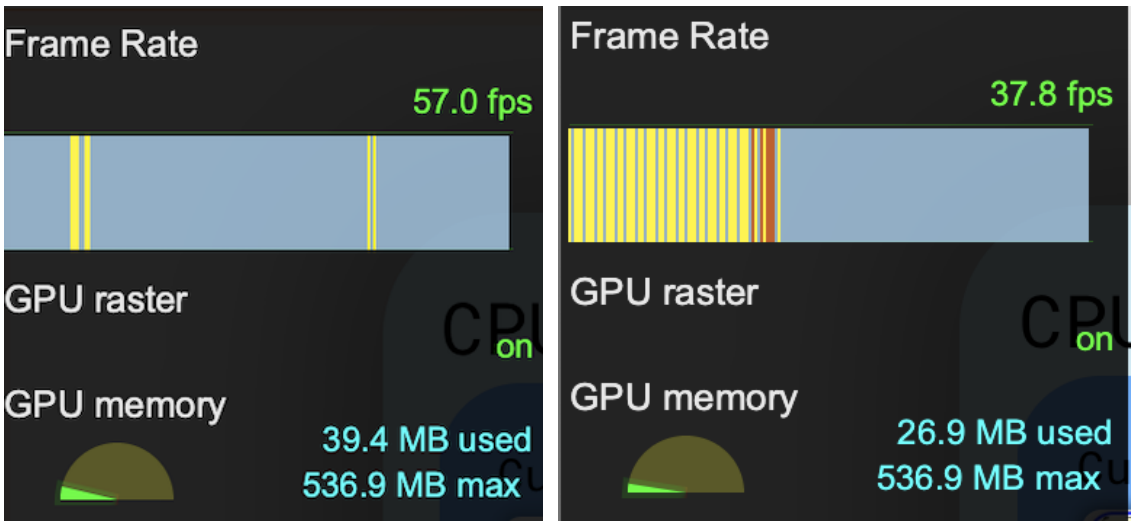


Abbildung I 4 Chrome DevTools Frame Rendering Stats während einer Animation

Ergebnis Chrome Lighthouse Test

Um die Seitenladezeit und andere Diagnostiken zu analysieren, verwenden wir den Chrome Lighthouse Test. Wir stellen unsere Webseite als “index.html” Datei lokal mit einem Python “SimpleHTTPServer” zur Verfügung, da der Lighthouse Test die Webseite nicht über das “File” Schema testen kann. Der Test misst die Seitenladezeit, in dem er die Datenrate eines 3G-Mobiltelefonnetz simuliert. Es wird eine TCP Round Trip Time (RTT) von 40 ms und eine Datenrate von 10,240 kbps simuliert. Es spielt somit keine Rolle, ob unser Server lokal auf dem System läuft oder über das Internet kontaktiert wird.

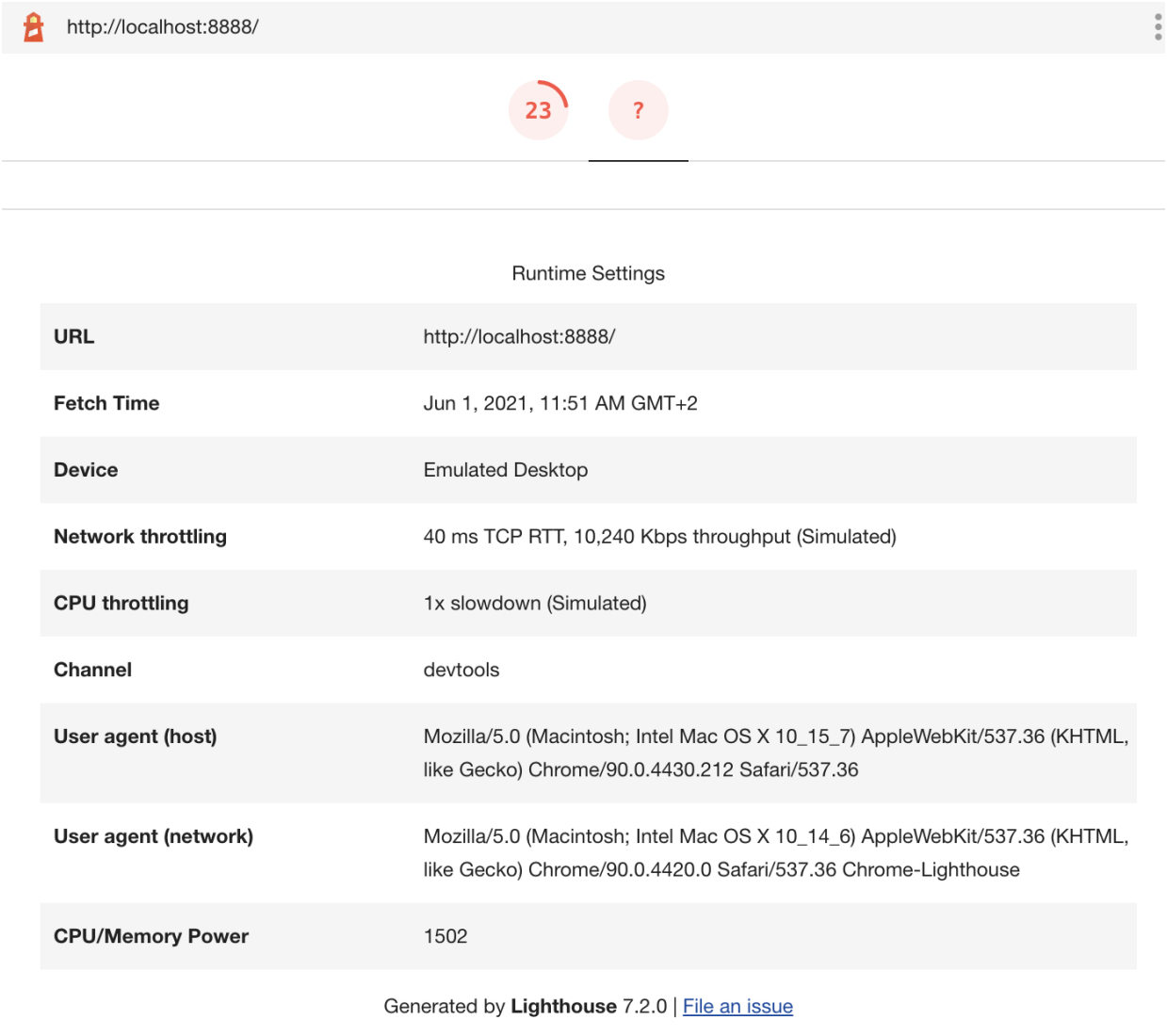


Abbildung I 5 Chrome DevTools Lighthouse Benchmark Runtime Settings

Seitenladezeit

Die Performancemessung von Lighthouse liefert 23 von möglichen 100 Punkten. Das Ergebnis ist in der Abbildung ersichtlich. Der grösste Einfluss auf dieses Resultat hat der “Speed Index” von 6.0 Sekunden, der die Seitenladezeit widerspiegelt. Der “First Contentful Paint” von 6.0 Sekunden zeigt, wie schnell das Grundlayout auf dem Bildschirm angezeigt wird.

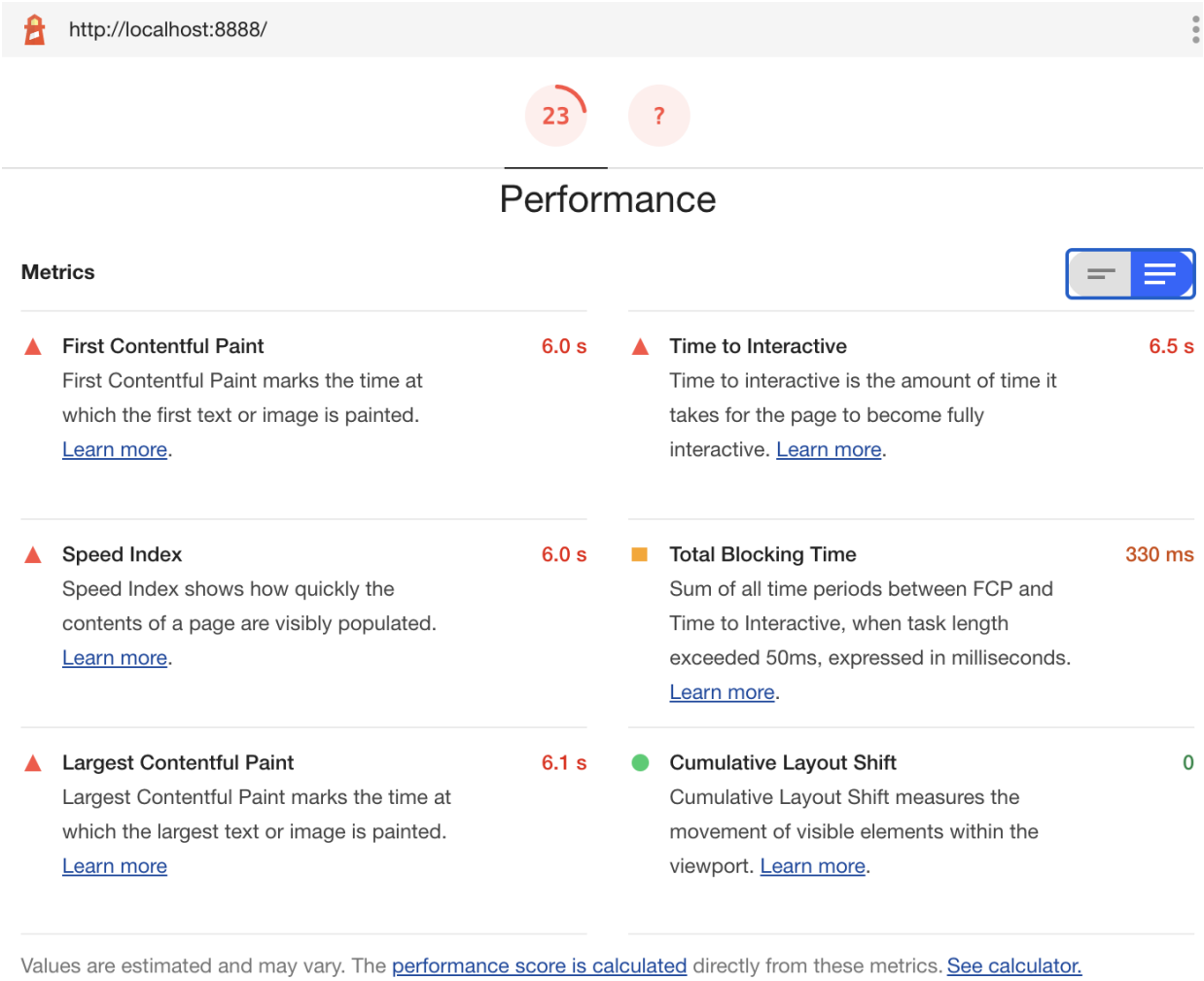


Abbildung I 6 Chrome DevTools Lighthouse Benchmark Performance Ergebnisse

Die “Time To Interactive” von 6.5 Sekunden zeigt, dass weitere 0.5 Sekunden vergehen, bis man mit der Webseite interagieren kann, um zum Beispiel den Assembly-Code im Editor zu ändern. Der “Cumulative Layout Shift” von 0 zeigt, dass sich das Layout der Webseite nicht verändert oder hin- und herspringt, sobald die Seite angezeigt wird, was unschön aussehen würde.

Weitere Resultate

Wie in der Abbildung zu sehen ist, zeigt der Lighthouse Test weiteres Verbesserungspotenzial der Webseite an. Da die ganze Webseite, wie gewollt, am Stück geladen wird. Der Test bemängelt die grosse Netzwerklast. Zusätzlich soll einiger JavaScript Code direkt nach dem Laden der Seite unbenutzt sein, da die nach JavaScript konvertierten C-Libraries erst beim der Benutzung des Simulators verwendet werden.

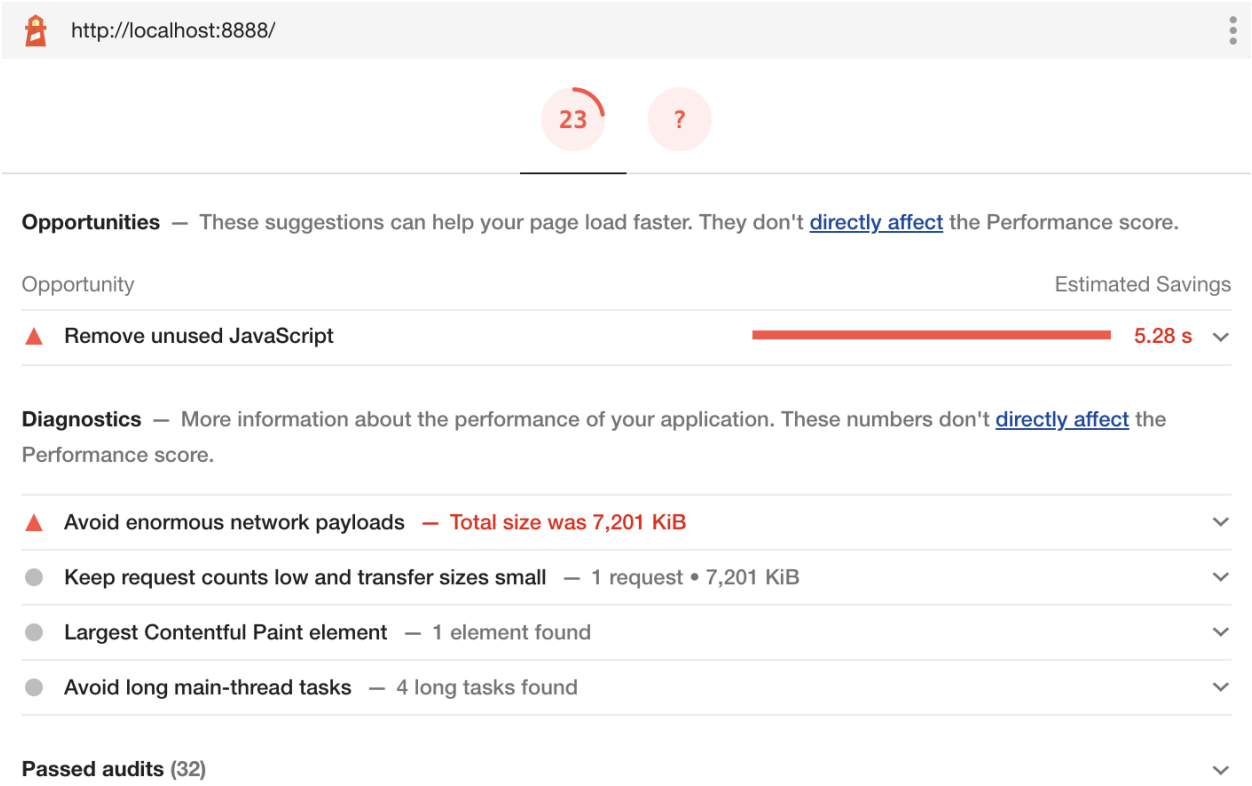


Abbildung I 7 Chrome DevTools Lighthouse Benchmark Performance Verbesserungsvorschläge

Anhang J Auswertung Usability Test 1

- Das Wichtigste ist verständlich, sogar für Personen, die fast keine Vorkenntnisse haben
- Nach einer halben Stunde ist der Prozessor Zyklus einigermassen klar, die Unterscheidung zwischen Execute und Get Instruction ist nicht ganz klar. Dies könnte durch CPU Box Design verbessert werden
- Simulator nach 5 Minuten bedienbar (ausser zum Code Editor zurückgelangen)
- Es braucht einen Zurück zum Code Editor Button
- Es braucht ein Change Log und der ursprünglich eingegebene Code
- Stack
    - Stack ist nicht verständlich, wenn man nicht weiss, was das ist.
    - es braucht Stack- und Base-Pointer analog zu Instruction Pointer
    - Es braucht ein Hover Text bei Pointer Bytes in Memory
    - Die Beispielprogramme sollten ohne push/pop sein, dafür sollte extra ein Programm namens Stack dabei sein. In den Programmen waren push rbp usw., das man beim Funktionsaufruf hinzufügt, dabei. Das könnte verwirrt
    - Falls man den Stack- und Base-Pointer, ähnlich wie der Instruction Pointer separat darstellt, sollte die Adresse in der gleichen Farbe wie Memory Adresse dargestellt werden.
- Code Editor:
    - Es ist nicht ganz einfach, den Code in den Editor zu kopieren.
    - Man sollte im Code Editor ausserhalb des Textfeldes klicken können, um hineinzuschreiben.
    - Das Beispielprogramm, dass bereits im Editor steht, muss Instruktionen unterschiedlicher Länge und mit unterschiedlichen Opcodes enthalten.
    - Editor sollte zuerst wachsen, bevor er Scrollbar wird
    - Der Code Editor sollte die Farbe rot nicht verwenden
- Instruktionen:
    - Unterscheidung [ ] und Immediate ist unklar
    - Jump Instruktionen müssen besser animiert werden (wahrscheinlich die Überschreibung des IP)
    - [rcx * 8 +0c7] Pointer Arithmetik ist unklar
    - Es braucht Beschreibung was EAX usw. ist
    - Es bräuchte ein Pop-up, das beschreibt, was CPU mit den Daten macht beim Ausführen.

Anhang K Auswertung Usability Test 2

Feedback von Usability Test 2

Hier werden Aussagen der Testpersonen aufgelistet, die sie von sich aus gemacht haben und nach denen wir nicht direkt gefragt haben. Beispielsweise haben zwei Personen gesagt, sie fänden das automatische Scrollen im Memory gut. Das bedeutet nicht, dass das den anderen drei Personen nicht gefallen hat, sondern dass sie das nicht speziell erwähnt haben.

Davis Simulator

https://kobzol.github.io/davis/

Probleme

- Man sieht nicht direkt, ob etwas passiert ist, Highlighting fehlt
- Es verändert sich nichts, wenn 0 geladen wird (2)
- Nutzen Memory unklar, wahrscheinlich, weil man nicht sieht, was bei mov ebx, [0x20] passiert, da dort 0 steht. (3)
- Wenn man nicht weiss, was Instruktionen machen sollen, versteht man es nicht
- Dezimalwert neben Registern unklar
- EIP nicht als Instruction Pointer erkannt und verwirrt, warum es hochgezählt wird
- Zu viele Informationen, man weiss nicht wo drücken
- Geht zu schnell, höhere Tickrate (3)
- Erste Erwartung bei Tickrate Slider ist, dass grösser = schneller
- Buttons nicht gut/verwirrend (3)
- Man sollte Step Button drücken können (2)
- Memory nicht angeschrieben mit Adressen
- Man sieht Prozessor Zyklus nicht

Positiv

- Code kann man einklappen
- Man sieht immer, wo man im Programm ist (2)
- Buttons, die man nicht nutzen soll, sind disabled
- Man sieht Register, Memory, macht Sinn, wenn man es bereits kennt (2)
- Breakpoints sind cool und intuitiv
- Slider, wenn man ihn noch langsamer einstellen könnte

Unser Simulator

Probleme:

- Im ersten Moment wird man überrascht von den vielen Informationen, nach kurzer Zeit findet man sich aber zurecht. (2)
- Toggle Tooltips sind nicht klar, ob sie aktuellen Zustand oder neuen Zustand anzeigen
    - besser: Animation for this step are enabled/disabled
- Menu Buttons sind prominenter als Editor Button, die Tooltips helfen jedoch, um den richtigen zu finden.
- Übersicht fehlt, die man konstant sehen kann, wo befinde ich mich im Programm (2)
- In Assembly Code-Box sieht man aktuelle Instruktion nicht
- Assembly Code-Box geht über Memory, was blöd ist, wenn man Code mit Memory Inhalt vergleichen will.
- Assembly Code Ansicht ungeeignet für grössere Programme (Syntax Highlighting fehlt)
    - z.B. Editor Komponente, die man nicht bearbeiten kann.
- Change Log wird zum Teil auch erwartet, dass Vorher/Nachher Werte stehen, oder auch die gelesenen. Ausserdem wird einmal IP-Wert zusätzlich erwartet. (3)
- CPU-Box: Wenn man nicht weiss, dass RSP ein Register ist, helfen die Tooltips beim Register RSP nicht.
    - Hover bei CPU-Box
- CPU-Box: Vorher/Nachher Vergleich, wenn gleicher Input wie Output eingefärbt, was hat sich geändert.
- Loading Spinner beim “Start Program” Button im Editor wurde zuerst nicht gesehen, besser wäre Spinner über ganze Seite.
    - Ausserdem müsste man theoretisch Browser Back Button animieren, wurde jedoch nie geklickt
- Bei klick/hover über RSP sollte Memory zu Stack scrollen und hover einstellen
- Es ist eher zum Lernen gedacht, als grosse Programme zu debuggen (2)
    - Breakpoints?
- Scrollbalken bei Registern soll nicht angezeigt werden, solange man nicht scrollen kann
- Erhöhung IP nicht klar
    - Evtl. Design von PowerPoint
- Animationen sollte man abbrechen können, wenn man sie bereits gestartet hat.
- Man soll Instruktionen als Ganzes überspringen können (3)
- Es ist nicht klar, wann Step Ausführung fertig ist (Animation ist fertig, aber Step noch eingeblendet)
- Immediate Byte wird für Fehler gehalten (2)
    - Hovering?
- Jump Instruction wird nicht richtig animiert, increment Instruction Pointer verwirrt hier. (2)
- Fehler Dialog fehlt, jedoch klar, dass Fehler passiert ist bei Reload. (3)
    - Besser schauen was wo abgefangen wird

- EBX unklar, weil es nicht unten angezeigt wird bei den Registern. Ist es dasselbe wie Register RBX, da die Werte an denselben Ort fliegen?
    - Reicht wohl im Editor, bei Simulation
    - evtl. pop up oder grosses Tooltip im Register
    - evtl. in Anleitung erklären
- Editor Ctrl + a wählt zum Teil ganze Seite aus, statt nur Code im Editor

Positives Feedback

- grosse Bereicherung, super, sehr cool, nice, macht Spass, macht Freude (5)
- intuitiv, verständlich, selbsterklärend, angenehm (5)
- Gut, dass man sieht, dass Flags geschrieben werden
- Animationen sehr gut, Move Animation sehr gut (3)
- Puls Animation, damit man sieht wo die Aktion passiert
- Stack gut dargestellt (4)
- Man sieht, dass Stack nicht gelöscht wird, sondern nur Pointer verschoben (3)
    - used Bytes helfen hier
- Darstellung CPU, Instruktionen gut
- Übersichtlicher als Linux Shell, gut zum Debuggen, löst Problem, dass man Register nicht ausgeben kann (3)
- Change Log sind praktisch
- Scrolling zum richtigen Ort (2)
- Fehleranzeige in Editor ist schön, gut dass man Ctrl-Z machen kann
- Unterscheidung CPU Cycle zu anderen Komponenten gut
- Prozessor Zyklus Schritte ersichtlich (3)
- Man sieht auf einen Blick, was zusammengehört
- Animationen ausschalten praktisch (2)

Verbesserungsideen

- Umwandlung Hex <-> Bit
- C reinladen und in Assembly verwandeln (ein Tool für ganze Vorlesung)
- Anleitung mit Theorie, Link zur Vorlesung
- Dark Theme
- Memory Bus
- Konsole unterstützen z.B. für Hello World Print Programm
    - Syscalls und Interrupt könnte man via Hook beim Unicorn abfangen und dann entweder leer lassen (so dass Programm nicht abbricht), Fehlerinfo anzeigen oder ausführen
- Tickrate einstellen können/schneller machen/langsamer machen 3
- Werte ins Memory schreiben können
    - evtl. bei Editor
- Bei Flags könnte man Hover mit Namen machen.
    - Flag Darstellung
- Alles am Stück laufen lassen
    - 1 Nein

- 1 Ja

Weitere Dinge, die wir beobachtet haben:

- Error Texte bei allen klar, die Programmiert haben
- Error Handling fehlt, trotzdem bei allen, bei denen Fehler aufgetreten ist, klar dass es ein Problem gibt.

Auswertung Fragen

Teilnehmer

- Matura:
  - 3 technische Berufsmatura
  - 2 wirtschaftliche Berufsmatura
- Lehre:
  - 2 Informatiker
  - 1 technische BM, aber keine Informatiklehre
  - 2 KV
- Bsys1 Kenntnisse:
  - 3 Gut
  - 2 Mittel

Vorinterview

- Erkläre kurz, wie der Prozessor arbeitet.
  - 1 Person beschreibt Prozessor-Zyklus vollständig
  - 2 Personen beschreiben Prozessor-Zyklus ohne Schritt Increment Instruction Pointer
  - 2 Personen beschreiben nur Execute Instruction Schritt

Davis Simulator

- Erkläre die Arbeitsweise des Prozessors
  - 4 Personen beschreiben, wie Prozessor sequenziell Programm abarbeitet
  - 1 Person sagt, dass sie den Prozessor-Zyklus hier nicht sehen kann
- Was passiert bei der Add-Instruktion und was ist darin alles involviert
  - Alle beschreiben Add-Instruktion, ohne die Flags zu erwähnen
  - Jemand erwähnt, dass die Add-Instruktion hier nicht so ersichtlich war, wie er sich das vorstellt

Unser Prozessor-Simulator

Erste 5 Minuten

- Erkläre die Arbeitsweise des Prozessors
  - Alle Personen beschreiben nun den Prozessor-Zyklus
- Was passiert bei der Add-Instruktion und was ist darin alles involviert
  - 4 Personen beschreiben die Add-Instruktionen mit Flags
  - 1 Person beschreibt ohne Flags, hat aber während der Animation die Flags gesehen und verstanden.
- Was gefällt dir an unserem Design besser und was weniger
  - Alle Personen fanden unser Design und die Bedienung unseres Simulators insgesamt besser und verständlicher
- Bedienung innert 5 Minuten
  - Alle haben das HTML-File, ohne zu zögern, aus dem Teams Chat heruntergeladen und im Google Chrome Browser innert Sekunden geöffnet.
  - 4 Personen haben die erste Aufgabe inkl. zurück zum Editor gehen innert 5 Minuten absolviert. Die meisten haben zu diesem Zeitpunkt bereits alle Knöpfe (Change Log, Assembly Code, Animation Toggles) bedient und verstanden und gingen zum Teil selbständig zurück zum Editor.
  - 1 Person brauchte 8 Minuten für die erste Aufgaben, da sie bereits ausführlich Feedback gegeben hat und dabei die Add-Animation verpasst hat. Die Person hat aber ohne Probleme das Programm neugestartet mit dem “Restart” Button.
  - Den Weg zurück in den Editor war für alle kein Problem.

Nach 20 Minuten

- Die meisten haben alle Buttons bereits während der ersten 20 Minuten selbständig verwendet
- Das Change Log wurde dabei weniger oft verwendet.
- Eine Person hat beim Hover über das Register RSP den Stack entdeckt und sofort das Stack Beispiel Programm gestartet, um zu sehen, wie der Stack funktioniert.
- Wir haben ihnen die Aufgaben gestellt, die sie noch nicht selbständig erledigt hatten und das Finden der Buttons und die Ausführung des Stack Programms war kein Problem
- Wir haben alle Personen bei der Ausführung einer Instruktion gefragt, was sie erwarten würden, was nun im Change Log steht
  - 2 alles, was sich geändert hat
  - 1 die gelesenen Werte
  - 1 Werte der In- und Outputs
  - 1 Vorher/Nachher Werte der Operanden
  - Alle haben die Assembly Interpretation der Instruktion erwartet und einer zudem noch der Wert des Instruction Pointers

    - Beim genaueren Betrachten der Werte war jedoch allen klar, dass es sich um die Werte handelt, die geändert wurden

Nachinterview

- Erkläre den Prozessor-Zyklus
    - Alle haben ihn korrekt mit allen 3 Schritten beschrieben
- Was für Daten befinden sich im Register RSP, wie führt der Prozessor Push/Pop aus
    - Alle haben den Stack verstanden und die Fragen richtig beantwortet
    - 1 Person hat zur Beantwortung der Frage, ob der Stackpointer erhöht oder dekrementiert wird, die Simulation zur Hilfe genommen
    - Dass der Stack immer mit 8 Bytes arbeitet, war nicht ersichtlich und wurde deshalb nicht richtig beantwortet
- Gefällt dir der Prozessor-Simulator
    - Alle finden ja
- Hilft der Prozessor-Simulator den Bsys1 Studierenden
    - Alle finden ja
- Hättest du den Simulator in der Übung verwendet?
    - 4 finden ja
    - 1 findet, er hätte ihn nicht verwendet, da er bereits genügend Vorkenntnisse hatte

Anhang L Protokolle Usability Test 1

Template Usability Test 1

Usability Test 1

Datum:
Testdurchführung:
Testperson:

Test wird durchgeführt auf [Branch name] Commit [Commit Hash]

Einschränkung: Die Demo Programme wurden als Textfile ausgehändigt und müssen von Hand reinkopiert werden. Jump und Call Instruktionen zeigen im Assembly Code falsche Adressen an, werden jedoch korrekt ausgeführt.

Vorinterview:

Beschreibung Person
Beschreibe die Arbeitsweise des Prozessors

Aufgabe 1:

Stell dir vor, du bist Student(in) im ersten Semester im Informatikstudium und erhälst dieses File (prozessor-simulator.html) in der Übung. Versuche zu verstehen, wie der Prozessor arbeitet.

Start:

-

Erster Durchlauf:

Programm: (Default Programm)
mov rax, [0x0]

mov [0x10], rax

- Get Instruction:
- Execute Instruction:
- Increment Instruction Pointer:

Aufgabe 2:

Du möchtest nun noch andere Programmausführungen sehen. Passe den auszuführenden Code an.
(Nur falls Personen nicht selbständig ausführen)

Aufgabe 3:

Starte das Multiplikationsprogramm.

Multiplikationsprogramm starten

-

Aufgabe 4:

Ändere das Programm zu folgendem:

```
mov rax, [0x0]
push rax
push 6
push 0xab
pop rbx
```

Code Editor

-

Programm ausführen (Stack)

-

Nachinterview:

Was zeigt ein Oval an?

-

Was zeigen die drei Felder oben an? (Schritte)

-

Was passiert im Schritt: Get Instruction?

-

Was passiert im Schritt: Execute Instruction?

-

Was passiert bei mov rax [0xa]

-

Was passiert im Schritt: Increment Instruction Pointer?

-

Verstehst du was das angemalte Byte ist (Stackpointer)?

-

Was stellen sich dir noch für Fragen?

-

Design Vorschlag “CPU Box”

Design Vorschlag CPU-Box und Change History vorführen.

Feedback:

-

Erkenntnisse aus diesem Test:

-

Usability Test 1

Datum: 26.04.2021
Testdurchführung: Eliane Schmidli
Testperson: S11

Test wird durchgeführt auf merged172and179 Branch, Commit 3560a65d9c276c3145fc4d68c06367d1e6b41b4d
(im Gegensatz zu master Branch, funktioniert hier Scrolling und es gibt Buttons für Demo Programme, die jedoch nicht funktionieren)

Einschränkung: Die Demo Programme wurden als Textfile ausgehändigt und müssen von Hand reinkopiert werden. Jump und Call Instruktionen zeigen im Assembly Code falsche Adressen an, werden jedoch korrekt ausgeführt.

Vorinterview:

S11 studiert Informatik an der OST. Zuvor hat sie das Gymnasium besucht. Sie hat das Modul Bsys1 im ersten Semester vor zweieinhalb Jahren besucht. Über den Prozessor-Zyklus weiss sie noch, dass darin verschiedene Komponenten, zum Beispiel Register involviert sind. Der Prozessor liest Daten hinein und hinaus und verarbeitet diese.

Aufgabe 1:

Stell dir vor, du bist Studentin im ersten Semester im Informatikstudium und erhälst dieses File (prozessor-simulator.html) in der Übung. Versuche zu verstehen, wie der Prozessor arbeitet.

Start:

- Startet Code Editor problemlos.
- Erkennt im Default Programm mov
- Drückt alle Knöpfe durch, um auszuprobieren, was passiert

Erster Durchlauf:

Programm: (Default Programm)

mov rax, [0x0]

mov [0x10], rax

- Get Instruction:
  - Versteht, das Instruktion geholt wird
- Increment Instruction Pointer
  - versteht sie nicht ganz.
  - wurde die Instruktion bereits reingelesen?
- Am Ende des Programms wird Instruktion ausgeführt, die an Stelle 0x10 kopiert wurde. Sie ist sich nun nicht sicher, ob endlos das ganze Memory ausgeführt wird.
  - Sucht nach Möglichkeit, auszusteigen
  - macht dann weiter und erkennt Reload Button

Zurück zum Code Editor

- Sie möchte zum Code Editor zurück, um ein anderes Programm auszuführen.
- Da sie keinen Knopf findet, startet sie das HTML File erneut.
- Sie wählt das Additionsprogramm aus und kopiert es in den Editor

Additionsprogramm

Programm: Additionsprogramm

- Versteht (erst jetzt), dass Instruction Pointer im Memory mit Rand um Byte angezeigt wird
- [rcx * 8 +0c7] verwirrt sie, sie meint, dass der Wert im RCX berechnet und reinge-schrieben wird.

Aufgabe 2:

Übersprungen, da sie selbständig ein Beispielprogramm gestartet hat.

Aufgabe 3:

Starte das Multiplikationsprogramm.

Multiplikationsprogramm starten

- öffnet erneut das HTML File
- startet problemlos das Programm
- Fragt sich (bei Push), was ESP ist.
- Fragt sich was cmp macht

- jnz: erkennt, dass es ein Jump sein könnte. Findet die Animation nicht klar (Problem: es ist nicht ersichtlich, dass IP verändert wird, deshalb kann sie wahrscheinlich nicht sagen, ob sie mit der Annahme Jump korrekt liegt)
- Öffnet den Programmcode, um nachzusehen, ob die Instruktion bereits vorgekommen ist.
- Wenn im Schritt “Execute Instruction” Daten in die CPU geholt werden, denkt sie manchmal plötzlich, dass das der Schritt “Get Instruction” war

Aufgabe 4:

Ändere das Programm zu folgendem:

```
mov rax, [0x0]
push rax
push 6
push 0xab
pop rbx
```

Code Editor

- Sie startet erneut das HTML File
- Sie gibt Code ein (ohne BITS 64)
- Sieht Fehlermeldung und realisiert sofort, dass sie ausversehen BITS 64 gelöscht hat

Programm ausführen (Stack)

- Sie erkennt, das register RSP involviert, aber weiss nicht wieso
- Erkennt, dass RBP dieselbe Adresse beinhaltet, wie das blaue Byte anzeigt
- fragt sich, ob Stackpointer Byte Rot ist, weil die Register rot dargestellt sind
- Erkennt das RSP und RBP Pointer sind
- Erkennt nicht, dass es sich um Stack handelt.

Nachinterview:

Was zeigt ein Oval an?

- Byte Code
- Versteht Adressierung

Was zeigen die drei Felder oben an? (Schritte)

- Schritte

Was passiert im Schritt: Get Instruction?

- Liest Funktion ein

- Ist sich unsicher, ob dort Daten hineingeladen werden, da es sie verunsichert, dass in Execute Instruction auch noch Bytes verschoben werden

Was passiert im Schritt: Execute Instruction?

- führt Instruktion aus und bewegt Daten

Was passiert bei mov rax [0xa]

- schiebt Daten ins rax
- Nicht sicher, ob Adresse oder Wert?

Was passiert im Schritt: Increment Instruction Pointer?

- Das Blaue Ding (Rand um Byte, wo IP drauf zeigt) wird neu gesetzt

Verstehst du was das angemalte Byte ist (Stackpointer)?

- Nein

Was stellen sich dir noch für Fragen?

- Was passiert mit eingelesenen Zahlen, die bei der CPU landen? Es bräuchte ein Po-pup das sagt, ich berechne die Adresse usw.

Design Vorschlag “CPU Box”

- Cpu box hilft, weil man dann weiss, wo Daten hingehen und sie nicht einfach ver-schwinden.
- Sie erkennt, dass CPU Daten einliest, verarbeitet und den Output wieder zurück-schreibt

Feedback:

- Es braucht ein Hover über dem Stackpointer Byte
- Es braucht einen Zurück zum Code Editor Button
- Changelog:
  - Es braucht einen Stop Button während der Animation
  - Man sollte einzelne Schritt noch einmal ausführen können
  - => Ich habe ihr Change Log Design Vorschlag gezeigt und sie fand, das würde auch reichen, damit man noch einmal nachlesen kann, was passiert ist.
- Man muss extra Code separat aufmachen, um zusehen was passiert ist.
- Es sieht sehr schön aus

Erkenntnisse aus diesem Test:

- Das Wichtigste ist verständlich
- Nach einer halben Stunde ist der Prozessor Zyklus einigermassen klar, die Unter-scheidung zwischen Execute und Get Instruction ist nicht ganz klar. Dies könnte durch CPU Box Design verbessert werden

- Es bräuchte ein Pop-up, das beschreibt, was CPU mit den Daten macht beim Ausführen.
- Stack ist nicht verständlich, es braucht Stack und Basepointer analog zu Instruction Pointer
- Es braucht ein Hover Text bei Pointer Bytes in Memory
- Die Beispielprogramme sollten ohne push/pop sein, dafür sollte extra ein Programm namens Stack dabei sein. In den Programmen waren push rbp usw., das man beim Funktionsaufruf hinzufügt, dabei. Das verwirrt
- Simulator nach 5 Minuten bedienbar (ausser zum Code Editor zurückgelangen)
- Es braucht einen Zurück zum Code Editor Button
- Es braucht ein Change Log und der ursprünglich eingegebene Code
- Unterscheidung [ ] und Immediate ist unklar
- Jump Instruktionen müssen besser animiert werden (wahrscheinlich die Überschreibung des IP)
- [rcx * 8 +0c7] Pointer Arithmetik ist unklar
- Es braucht Beschreibung was EAX usw. ist

Usability Test 1

Datum: 26.04.2021
Testdurchführung: Eliane Schmidli
Testperson: S12

Test wird durchgeführt auf merged172and179 Branch, Commit 3560a65d9c276c3145fc4d68c06367d1e6b41b4d
(im Gegensatz zu master Branch, funktioniert hier Scrolling und es gibt Buttons für Demo Programme, die jedoch nicht funktionieren)

Einschränkung: Die Demo Programme wurden als Textfile ausgehändigt und müssen von Hand reinkopiert werden. Jump und Call Instruktionen zeigen im Assembly Code falsche Adressen an, werden jedoch korrekt ausgeführt.

Vorinterview:

S12 studiert Informatik an der OST. Zuvor hat er eine Lehre als Polymechaniker gemacht und in der Elektronik gearbeitet. Er hat das Modul Bsys1 im ersten Semester vor dreieinhalb Jahren besucht. Er beschreibt CPU sehr detailliert. Er erwähnt, dass CPU rechnet und Register hat und den Takt für die Ausführung angibt.

Aufgabe 1:

Stell dir vor, du bist Studentin im ersten Semester im Informatikstudium und erhälst dieses File (prozessor-simulator.html) in der Übung. Versuche zu verstehen, wie der Prozessor arbeitet.

Start:

- Startet Code Editor problemlos.
- Erkennt, dass es sich um Assembly handelt
- Startet Simulator

Erster Durchlauf:

Programm: (Default Programm)

mov rax, [0x0]

mov [0x10], rax

- Scrollt Memory herunter und sagt, dass er im Memory einen Stack erwarten würde. Sieht während dem sprechen das Byte, das Stackpointer darstellen soll, und fragt, was das sein soll.
- Get Instruction
    - Fragt sich kurz, was im Memory für Daten sind und erkennt dann, dass es sich um Programmcode handelt.
- Execute Instruction
    - Ist sich anhand vom Assembly nicht sicher, ob [0x0] Position 0 oder Wert 0 bedeutet.
    - Nach Animation klar
- Increment Instruction Pointer
    - Erkennt, dass er um 8 Byte erhöht wird.

Zurück zum Code Editor

- Er möchte zum Code Editor zurück, um ein anderes Programm auszuführen.
- Da er keinen Knopf findet, bearbeitet er den Link in der Adresszeile
- Er löscht Code im Editor und möchte Switchprogramm hinein kopieren.
- Er erkennt, dass es einen feinen Rahmen, um das Textinputfeld gibt und findet es doof, dass man nicht ausserhalb des Feldes klicken kann, um reinzuschreiben.
- Bei einem grösseren Programm fände er es gut, wenn der Editor grösser wird und nicht so früh ins Scrollen übergeht.

Switch Variables Programm

- Als er das Programm startet, findet er, es fehle eine Ladeanimation auf dem Button und beim Laden des Simulators.
- Er möchte bei Bytes in Current Instruction den Opcode und die Operanden hovern können.
- Er versteht zwar, dass die Daten in die CPU kommen und die dann damit arbeitet, findet es aber verwirrend, dass die Bytes verschwinden.

Aufgabe 2:

Übersprungen, da sie selbständig ein Beispielprogramm gestartet hat.

Aufgabe 3:

Starte das Multiplikationsprogramm.

Multiplikationsprogramm starten

- Er startet Editor wieder über die Adresszeile
- Bei Ausführung Push rbp ist er verwirrt und fragt sich warum Daten soweit unten ins Memory geschrieben werden
- Erkennt dann plötzlich, das Stack beim farbigen Byte anfängt.
- Erkennt bei mov eax, dass es sich, um die Hälfte des Registers RAX handelt
- Er versteht beim ersten Loop Durchlauf bereits, was passieren soll im Multiplikaitons-programm

Aufgabe 4:

Ändere das Programm zu folgendem:

```
mov rax, [0x0]
push rax
push 6
push 0xab
pop rbx
```

Code Editor

- Er startet ihn über die Adresszeile
- Er löscht alles bis auf BITS 64 und gibt Code ein.

Programm ausführen (Stack)

- Er erkennt nun Zusammenhang zwischen RSP Register und Stackpointer. Ist jedoch unsicher, was RBP ist.

Nachinterview:

Was zeigt ein Oval an?

- 1 Byte Speicher

Was zeigen die drei Felder oben an? (Schritte)

- 1 Prozessor-Zyklus

Was passiert im Schritt: Get Instruction?

- liest Befehl an Position des IP

Was passiert im Schritt: Execute Instruction?

- CPU führt Instruction aus

Was passiert bei mov rax [0xa]

- schiebt Daten ins rax

- Versteht Unterschied 0xa und [0xa]

Was passiert im Schritt: Increment Instruction Pointer?

- Befehlslänge wird zu IP hinzugefügt

Verstehst du was das angemalte Byte ist (Stackpointer)?

- RSP zeigt dort drauf, Stackpointer
- Blau = heap? (Wäre Basepointer)

Was stellen sich dir noch für Fragen?

- Warum bewegen sich bei Flag Write soviele Daten (bei cmp wird gerechnet und deshalb alle gesetzt, geplant wäre jedoch, dass sie alle miteinander gesetzt werden)

Design Vorschlag “CPU Box”

- Cpu box hilft, weil Daten nicht einfach verschwinden.
- Etwas übertrieben für mov Instruktion, die könnte ja direkt zum RAX fliegen.
- Er erkennt, dass CPU Daten einliest, verarbeitet und den Output wieder zurückschreibt

Feedback:

- Tooltipps bei Opcodes
- Gefällt, würde er benutzen
- Es braucht Buttons:
    - Home Button zum Editor
    - Zum ersten Schritt zurück
    - Einen Schritt zurück
    - Ich zeige ihm Design für Change Log. Er findet das in Ordnung als Ersatz für “Zurück Button”

Erkenntnisse aus diesem Test:

- Das Wichtigste ist verständlich
- Nach kurzer Zeit ist Prozessor-Zyklus klar
- Stack ist nicht verständlich, es braucht Stack und Basepointer analog zu Instruction Pointer
- Die Beispielprogramme sollten ohne push/pop sein, dafür sollte extra ein Programm namens Stack dabei sein. In den Programmen waren push rbp usw., das man beim Funktionsaufruf hinzufügt, dabei. Das verwirrt.
- Simulator nach 5 Minuten bedienbar (ausser zum Code Editor zurückgelangen)
- Es braucht einen Zurück zum Code Editor Button
- Es braucht ein Change Log und der ursprünglich eingegebene Code

- Man sollte im Code Editor ausserhalb des Textfeldes klicken können, um hineinzuschreiben.
- Editor sollte zuerst wachsen, bevor er Scrollbar wird.

Usability Test 1

Datum: 25.04.2021
Testdurchführung: Eliane Schmidli
Testperson: S13

Test wird durchgeführt auf master Branch, Commit
8f4159fec5039f94e6671cf8de21aa81edce68b0

Einschränkung: Die Demo Programme wurden als Textfile ausgehändigt und müssen von Hand reinkopiert werden. Jump und Call Instruktionen zeigen im Assembly Code falsche Adressen an, werden jedoch korrekt ausgeführt. Automatisches Scrolling funktioniert nicht sauber.

Vorinterview:

S13 studiert Wirtschaft. Seine Informatikkenntnisse beschränken sich auf ein Modul, dass er im Rahmen des Wirtschaftsstudiums absolviert hat. Er kann die Funktionsweise eines Prozessors nicht beschreiben. RAM ist ein Begriff für ihn und er stellt sich das als eine Art Tabelle vor, wo man durch Angabe der Zeilen und Spalten darauf zugreifen kann.

Test:

Aufgabe 1:

Stell dir vor, du bist Student im ersten Semester im Informatikstudium und erhälst dieses File (prozessor-simulator.html) in der Übung. Versuche zu verstehen, wie der Prozessor arbeitet.

Start:

- Startet Code Editor und Simulator problemlos.
- Denkt auf den ersten Blick, dass man in Bytes reinschreiben kann

Erster Durchlauf:

Programm: (Default Programm)

mov rax, [0x0]

mov [0x10], rax

- Get Instruction:
  - erkennt Zusammenhang Maschinen Bytes und Assembly Code innerhalb Current Instruction
  - Versteht Schritt Get Instruction
- Execute Instruction:
  - Versteht, dass mov für move steht.
  - Erkennt, dass CPU involviert
  - Operanden sind noch unklar
- Increase Instruction Pointer:
  - Versteht er nicht
  - frag sich, ob das gelbe Feld (=Adresse) im Instruction Pointer und gelbe Felder bei Memory (Adresse 1. Zeile) zusammenhängen
- startet zweiter Durchlauf indem er auf “Reload” Button klickt

Zweiter Durchlauf

Programm: (Default Programm)

mov rax, [0x0]

mov [0x10], rax

- Execute Instruction
  - Versucht Funktionsausführung zu verstehen.
  - erkennt Operand RAX
  - erkennt, dass je nach Anordnung der Operanden, die Daten in die andere Richtung verschoben werden.
  - Fragt sich warum genau 8 ovale verschoben werden und ob Operand [0x000] damit zu tun hat
- Increment Instruction Pointer
  - Erkennt Zusammenhang Blauer Rahmen um Byte in Memory und Instruction Pointer Adresse
  - Merkt, dass IP immer um 8 erhöht wird und fragt sich warum
- Sieht Farbiges oval weiter unten im Memory (= Stack Pointer) und fragt sich, ob das ein Fehler ist.

Aufgabe 2:

Du möchtest nun noch andere Programmausführungen sehen. Passe den auszuführenden Code an.

Zurück zum Editor

- Versucht, ob er die Bytes irgendwo im Simulator ändern kann.
- Ist etwas verwirrt (hat wahrscheinlich vergessen, dass er als erstes einen Code Editor gesehen hat, bevor er den Simulator gestartet hat, es fehlt offensichtlich ein Knopf mit dem man zurück zum Editor kommt)
- Startet html File neu.

Bearbeitung Code im Editor

- Möchte nun Code im Editor anpassen.
- Versucht [0x0] auf [10x10] zu ändern
- Bemerkt die Fehlermeldung und die Angabe der Zeile, wo der Fehler liegt.
- Versucht verzweifelt die Zahl zu bearbeiten
- Wenn 0x voraus steht wird die Zahl rot (weil korrekt), er denkt jedoch das stehe für falsch.
- Geht davon aus dass “0x0” für “Spalte x Zeile” in Memory steht (er hat im Vorinterview angegeben, dass er sich Memory als Tabelle vorstellt)
- Ich greife ein und erkläre, dass 0x für Hexadezimal Zahl steht.

Startet modifiziertes Programm

Programm: (Default Programm modifiziert)

mov rax, [0x10]

mov [0x10], rax

- Versucht Increment Instruction Pointer zu verstehen
- Ihn verwirrt, dass diese ovale in den gelben Bereich hineinfliegen (Adresse in IP)
- Bemerkt, dass er Simulator neustarten kann, indem er einen Reload im Browser macht.
- Er versteht nicht, warum bei Get Instruction 8 ovale geholt haben und überlegt sich, ob 48 (Opcode) markiert, dass hier die nächste Instruktion liegt, oder ob das immer 8 ovale sind.

Aufgabe 3:

Starte das Multiplikationsprogramm.

Programm in Editor hineinkopieren

- Klappt erst beim zweiten Ansatz (beim markieren des Codes im Editor gerät man schnell aus dem Textfeld hinaus und die Bearbeitung ist nicht möglich)

Multiplikationsprogramm starte

Programm: Multiplikationsprogram aus Vorlage

- Instruktion hat nun plötzlich nicht mehr 8 Ovale
- Erkennt nun, dass der Instruction Pointer genau um die Anzahl Ovale erhöht wird, die Instruktion lang war. Versteht nun also die Animation Increase Instruction Pointer.
- push rbp verwirrt ihn komplett, da mehrere Register gelesen werden
- Er versteht nun, dass Operand [0x…] Adresse im Memory ist. Ist dann jedoch verwirrt als dann bei Immediate 0x… die Animation anders ist (erkennt nicht, dass [ ] einen Unterschied machen)
- Versteht Zusammenhang eax und rax nicht, trotz Animation nicht

Aufgabe 4:

Ändere das Programm zu Stack programm: Wurde weggelassen, das der Proband nicht weiss was der Stack ist und die Zeit (30 Minuten) um war.

Nachinterview:

Was zeigt ein Oval an?

- Daten in binär
- Versteht auch Adressierung der Bytes

Was zeigen die drei Felder oben an? (Schritte)

- Schritt in dem man sich befindet

Was passiert im Schritt: Get Instruction?

- CPU holt gewisse Anzahl Ovale aus Memory, ab dem Instruction Pointer

Was passiert im Schritt: Execute Instruction?

- CPU macht das, was bei Current Instruction (Assembly Code) steht

Was passiert bei mov rax [0xa]

- schiebt etwas ins rax
- zweiter Operand ist nicht ganz klar, könnte Oval sein, wo die Daten stehen

Was passiert im Schritt: Increment Instruction Pointer?

- Wert von Instruction Pointer wird um Anzahl Ovale erhöht, die bei Current Instruction stehen

Verstehst du was das angemalte Byte ist (Stackpointer)?

- Nein

Weisst du was ein Register ist?

- Beschreibt Memory
- Sieht bei nächster Frage plötzlich, dass Register Block oben angeschrieben ist.
- Korrigiert Antwort auf: Zwischenspeicher, Teil von der CPU

Was stellen sich dir noch für Fragen?

- Die Instruktionen (z.B. dec, push) sind noch nicht alle klar
- Flags sind unklar

Design Vorschlag “CPU Box”

- Findet Animation verständlich
- Erkennt, dass CPU Daten einliest, verarbeitet und den Output wieder zurückschreibt

Feedback:

- Sieht nützlich aus
- Ist bestimmt für Unterricht geeignet
- sieht optisch gut aus
- es ist hilfreich, dass Instruction Pointer umrandet ist.

Erkenntnisse aus diesem Test:

- Das Wichtigste ist verständlich, sogar für Personen, die fast keine Vorkenntnisse haben
- Nach einer halben Stunde ist der Prozessor Zyklus klar, einige Instruktionen (z.B. mov, add) sind klar
- Stack ist nicht verständlich, wenn man nicht weiss, was das ist.
- Das Beispielprogramm, dass bereits im Editor steht, muss Instruktionen unterschiedlicher Länge und mit unterschiedlichen Opcodes enthalten.
- Die Beispielprogramme sollten ohne push/pop sein, dafür sollte extra ein Programm namens Stack dabei sein. In den Programmen waren push rbp usw., das man beim Funktionsaufruf hinzufügt, dabei. Das könnte verwirrt
- Simulator nach 5 Minuten bedienbar (ausser zum Code Editor zurückgelangen)
- Es braucht einen Zurück zum Code Editor Button
- Es ist nicht ganz einfach, den Code in den Editor zu kopieren.
- Der Code Editor sollte die Farbe rot nicht verwenden
- Falls man den Stack- und Basepointer, ähnlich wie der Instruction Pointer separat darstellt, sollte die Adresse in der gleichen Farbe wie Memory Adresse dargestellt werden.
- Unterscheidung [ ] und Immediate ist unklar
- Es braucht Beschreibung was EAX usw. ist

Anhang M Protokolle Usability Test 2

Template Usability Test 2

Usability Test 2

Name:

Datum:

Der Test wurde durchgeführt auf dem Branch 112Layout Commit:

ee46804aea14c73988a0b85719a919551a6894e4

Testpersonen informieren:

- dass Aufzeichnung gemacht wird
- dass Sie anonymisiert werden
- dass sie laut denken müssen
- dass sie Fragen stellen sollen während dem Test, die am Schluss beantwortet werden.

Vorinterview

- Name:
- Alter:
- Studiengang und Semester:
- Ausbildung vor Studium:
  - gymnasiale Maturität
  - technische Berufsmaturität
  - wirtschaftliche Berufsmaturität
  - andere?
  - und Name der Lehre
- Schätze deine Kenntnisse über Bsys1
- Erkläre kurz wie der Prozessor arbeitet.

Test

- Zeit stoppen!

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:

```
section .text
   MOV EBX, [0x20]
   INC EAX
   ADD EAX, EBX
```

- Ca. nach 4 Minuten abbrechen und folgende Fragen stellen:
- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
- Gefällt dir das Design:
  - Was besonders
  - Was nicht

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:

```
BITS 64
mov ebx, [0x20]
inc eax
add eax, ebx
```

- Ca. nach 4 Minuten abbrechen und folgende Fragen stellen:
- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
- Gefällt dir das Design:
  - Was gefällt dir besser als beim Vorherigen
  - Was weniger

Prozessor-Simulator 2. Runde

- Ändere nun das Programm

(Nicht die Worte “zurück” oder “Editor” verwenden. Schauen ob Person innert 1 Minute den Weg zurück in den Editor findet)

- Du darfst nun den Simulator frei ausprobieren

- Nach 15 Minuten (bzw. die letzten 10 Minuten) folgende Aufgaben stellen, falls nicht bereit gelöst
    - Führe nun das Beispielprogramm “Stack” aus
    - Was machst du wenn du gewisse Animationen verstanden hast und diese überspringen möchtest, zum Beispiel wie die Instruktion eingelesen wird.
    - Gibt es eine Möglichkeit die Vergangenen Schritte des Programms zu sehen?
    - Schaue nach, welchen Code du eingegeben hast.

Nachinterview

- Fragen wiederholen, die vorher falsch beantwortet wurden.
- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
- Was ist ein Register
- Was für Daten befinden sich im Register RSP
- Was ist das Farbige Byte im Memory
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
- Wie hat dir der Simulator gefallen
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verstehen?
- Hättest du den Simulator gerne in den Übungen verwendet?
- Gab es Dinge die dich verwirrt haben?
- Was wünschst du dir, was fehlt noch
- Feedback zum Usability Test
- Fragen wegen Kredits

Hinweis, dass sie das Html File löschen müssen wegen der Lizenz

Usability Test 2

Name: S21

Datum: 12. Mai 2021

Vorinterview

- Studiengang und Semester: Informatik, 2. Semester
- Ausbildung vor Studium:
  - KV, wirtschaftliche BMS
- Schätze deine Kenntnisse über Bsys1: Hält sich in Grenzen
- Erkläre kurz wie der Prozessor arbeitet. :
  - Transistoren
  - sehr einfache arithmetische Operationen
  - Instruktion vom Prozessor ausgeführt, von Memory nach Prozessor oder umgekehrt
  - Führt mithilfe der Register die Operationen aus
  - Instruktionen sind je nach Betriebssystem 64 Byte/Bit gross
  - zwischen Memory und Register ist cash

Fragt nach Ziel unseres Programms

Test

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:

```
section .text
    MOV EBX, [0x20]
    INC EAX
    ADD EAX, EBX
```

Wert wird in EBX geladen (nicht sicher was)

EAX wird erhöht

Sieht im Simulator Register und Flags und Memory

Klickt auf Ascii Interpretation

Klickt Start

Ist verunsichert, weil er nicht sieht das bei play etwas passiert ist

Merkt dann, dass im Register etwas passiert ist

Ist verunsichert, weil er erwartet, das Wert ins Register geladen wird, aber nichts passiert ist.

Bemerkt, dass nichts passiert, weil 0 ins EBX geladen wird.

Ist verunsichert weil sich bei Addition nichts im EBX ändert

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - beinhaltet zwei Register, und schreibt ins Register Eax
    - erwähnt Flags nicht
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben Gesehenen erklären?
    - oben gestartet und sequenziell durchgelaufen
- Gefällt dir das Design:
    - Was besonders
        -
    - Was nicht
        - zu viele Infos, nicht klar wo schauen
        - erwarte zum Beispiel ein Highlighting
        - man sieht Änderungen erst nach 3 Mal drücken

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:

```
BITS 64
mov ebx, [0x20]
inc eax
add eax, ebx
```

Startet mit Doppelklick Program

Versucht Programm zu verstehen:

- nicht sicher was ‘BITS 64’ ist
- mov von Speicherplatz 20
- eax erhöhen, um eins
- eax und ebx addieren und ins eax schreiben

Klickt Start Program

“ou”

man muss sich zuerst zurechtfinden
erkennt CPU, Memory etc.
ist skip drin? (Tooltips bei Toggles sind unklar)
klickt Next Step
erkennt Instruction lesen
klickt Next Step
liest von Memory und schreibt in RBX (statt EBX)
klickt Next Step
nicht klar ob Increment Instruction Pointer bereits Increment war.
nicht klar in welchem Schritt man ist (Instruktion)
er möchte das gesamte Programm als Übersicht, im anderen Simulator hat man dieses direkt daneben gesehen. Bei uns sieht man immer nur 1 Zeile.
er nimmt an, dass man das 1 bis 2 Mal machen muss und es dann versteht
Eliane fragt nach add Instruktion (4 Minuten waren um), er hat diese jedoch nicht genau angeschaut, weil er Feedback zum Simulator gegeben hat. Er startet das Programm noch einmal neu.
(reload Wartezeit ist zu lange) er war sich unsicher, ob es läuft
klickt alles nochmals durch ohne Skip

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - hat Flags nun erwähnt, Instruktion wird vom Memory geladen, erwähnt das RAX und RBX involviert sind, nur ersten 4 Bytes werden geschrieben da EBX
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - bereits vorher beantwortet.
- Gefällt dir das Design:
    - Was gefällt dir besser als beim Vorherigen
        - Gesamtorientierung war besser
    - Was weniger
        - Ich möchte das Programm als Übersicht sehen, und erkennen in welchem Schritt ich bin

Prozessor-Simulator 2. Runde

- Ändere nun das Programm

Er hovert über Menübuttons, da diese prominenter sind, liest den Hover Text und gelangt schlussendlich zum Editor Button

-> Er hat 8 Minuten gebraucht für den Durchlauf, rechnet man die Zeit für die Wiederholung des Programms weg, würden 5 Minuten genügen

- Du darfst nun den Simulator frei ausprobieren

klickt Swap Beispielprogramm und schaut es an

- sagt, dass man bei first var und second zwei Adressen hat
- Verwirrt von global start und _start
- von Speicherort firstVar nach rax laden

Startet das Program

sucht CAFE und first var im Memory

blöd, dass Assembly Code Fenster das Memory überdeckt

hat gefunden

nicht sicher, ob 0x1337 von first var Wert oder Adresse ist, wo Wert von 1337 liegt

Sagt, dass Instruction Pointer um 8 erhöht wird, da RAX involviert ist

- Führe nun das Beispielprogramm “Stack” aus
    - hat direkt gefunden
    - Versucht Programm zu interpretieren
        - zuerst Instruction auf den Stack
        - dann Wert auf den Stack
        - nicht sicher, ob mov etwas mit Stack zu tun hat
        - RAX Wert auf den Stack
        - dann nach rbx schreiben
        - und dann nochmals pop nach Speicher
    - Startet Simulator
    - Erinnert sich nicht was RSP ist in Execution Box und versucht hover über Execution Box bei Rsp (weiss nicht, dass es ein Register ist und schaut deshalb nicht unten nach)
    - ahh ist das dort, wo Stack angelegt ist im Speicher
    - nicht klar von wo quad word Werte kommen
    - ah stimmt stack geht hinauf, also in die andere richtung
    - ich mach mal ein bisschen schneller ... (klickt aber nicht auf Skip)
    - jetzt brauchen wir Stack Pointer, um herauszufinden, wo Wert hingeschrieben werden soll
    - Wert bei Stack Pointer laden und in Rbx schreiben
- Was machst du wenn du gewisse Animationen verstanden hast und diese überspringen möchtest, zum Beispiel wie die Instruktion eingelesen wird.
    - findet Skip Toggles
- Gibt es eine Möglichkeit die Vergangenen Schritte des Programms zu sehen?
    - findet Change Log
    - sieht Vergange Instruktion

- Speicherplatz und Wert
- je nachdem, ob vom Speicher oder Register nimmt
- Meint, es zeige die gelesenen Werte.
- nimmt an, dass Nummer die Zahl der Schritte bedeuten

Nachinterview

- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
  - nimmt Instruktion von Memory
  - führt diese aus
  - kalkuliert Instruction Pointer Position
  - Instruktion wird um Länge der instruktion erhöht, die Länge der Instruktion variiert je nach Grösse der Parameter (RAX -> 8 Bytes)
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
  - nimmt Wert von dieser Adresse
- Was ist ein Register
  - Wurde bereits im Vorinterview richtig gesagt.
- Was für Daten befinden sich im Register RSP
  - ist glaube ich Stack Pointer
  - bei push wird RSP verringert und bei pop erhöht
  - immer um 8 Bytes erhöht
  - Findet nun heraus, dass man über RSP hätte hovern können
- Was ist das Farbige Byte im Memory
  - bereits beantwortet
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
  - bereits beantwortet
- Wie hat dir der Simulator gefallen
  - findet Design gut
  - Stack gut dargestellt
  - Instruktionen und CPU gut
  - wünscht sich bessere Animation bei Erhöhung von Instruction Pointer, damit es klar ist warum erhöht
  - fragt, ob CPU Input Werte bei Ausführung nochmals in eine Art Register geladen wird
  - wünscht sich besseren Überblick über aktuelles Programm, z.B. für jump
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verstehen?
  - ja auf jeden Fall
- Hättest du den Simulator gerne in den Übungen verwendet?
  - ja
  - in Linux Shell ist es viel unübersichtlicher
  - sanfter an das Thema herangeführt
- Gab es Dinge die dich verwirrt haben?
- Was wünschst du dir, was fehlt noch

    - man könnte mega coole Sachen machen z.B. Umwandlung in Bits
    - man könnte z.B. C reinladen und dann nach Assembly konvertieren
    - Es bräuchte eine Anleitung z.B. Text und zusätzliche Theorie, z.B. um was wird Instruction Pointer erhöht. Link zu Vorlesungsunterlagen
- Feedback zum Usability Test
    - fand es ganz ok
- Fragen wegen Kredits
    - ja gerne

Usability Test 2

Name: S22

Datum: 12. Mai 2021

Vorinterview

- Studiengang und Semester:
- Ausbildung vor Studium:
  - -> informatiker als lehre
- Schätze deine Kenntnisse über Bsys1
  - mehr oder weniger präsent
  - hart gefühl konnte Vorlesungen folgen
- Erkläre kurz wie der Prozessor arbeitet.
  - Programm gelangt von Festplatte ins RAM
  - werden von Prozessor interpretiert
  - dann sequentielles Abarbeiten
  - Prozessor hat Caches
  - es gibt Instruction Pointer, der sagt wo nächte Instruktion liegt

Test

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:

```
section .text
    MOV EBX, [0x20]
    INC EAX
    ADD EAX, EBX
```

verwirrt, dass nur einzelne Zahlen in Register

sieht Register, die sich ändern, wenn Play gedrückt

geht zu schnell

sagt es macht Sinn

klickt auf Memory Byte Size Buttons und weiss nicht was damit anfangen

wäre schön wenn er Step brauchen könnte (Step Button ist disabled)

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - EBX zu Wert EAX addiert und in EAX geschrieben
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - man hat nicht gesehen, ob sich etwas geändert hat, war zu schnell
    - beschreibt das Programm
- Gefällt dir das Design: Ja eigentlich schön
    - Was besonders
        - gut dass man Code, den man nicht braucht einklappen kann
        - dass Buttons, die man nicht nutzen soll, disabled werden
    - Was nicht
        - man möchte vielleicht nicht, dass es automatisch durchläuft
        - sieht Nutzen nicht von Memory

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:

```
BITS 64
mov ebx, [0x20]
inc eax
add eax, ebx
```

öffnet mit Doppelklick direkt im Browser

lacht beim Simulator öffnen

ist kurz überfordert

findet Next Step

sehr gut, dass man sieht, wo was passiert

findet Instruction Pointer

Man sieht Bytes die interpretiert werden

gefällt sehr gut

sagt genau, was mov mit Register macht

findet sehr gut, dass man Flags sieht

findet Base und Stack Pointer Register

klickt alles schnell durch und schaut zu

findet Skip und findet praktisch

finder History und findet praktisch

geht selbständig zurück zum Editor

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - Instruktion wird geladen
    - updated Register und Flags
    - dann Instruction Pointer erhöhen
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand vom gerade eben gesehenen erklären?
    - Bereits vorher beantwortet
- Gefällt dir das Design:
    - Was gefällt dir besser als beim Vorherigen
        - man sieht, was der Prozessor überhaupt macht, beispielsweise Interpretation Instruktion und Lesen und Schreiben der Werte
        - findet Puls Animation gut bei jeder Aktion, um zu sehen, was involviert ist
        - findet Change Log sehr gut
    - Was weniger
        - schwieriger, dass man zuerst auf Assembly Code klicken muss, um einen Überblick über das Programm zu erhalten -> findet aber zweitrangig

Prozessor-Simulator 2. Runde

- Ändere nun das Programm
    - Bereits von alleine erledigt.
- Du darfst nun den Simulator frei ausprobieren

hat gesagt, dass er beim Editor Loading Spinner nicht gesehen hat ursprünglich

hat Animation abgestellt

findet gut, dass man sieht bei welchem Step man ist

macht Hover über alle Base/Stack Pointer Sachen

klickt auf Assembly und Change Log Button

ahhh man sieht sogar Flags im Change Log

wäre schön Animation abzubrechen, wenn sie schon laufen

startet Stack Programm selbständig

findet super, dass es an den richtigen Ort scrollt

sagt findet super 3-mal

erwartet, dass man mit Doppelklick auf Memory Adresse im RSP zum Memory springen/scrollen kann

fragt sich, ob es hilfreich ist, wenn mal alles am Stück laufen lassen könnte, aber denkt eher nicht

wenn man alle Animationen abstellt, ist es schwierig zu Folgen
sagt das Assembly Code Button Ansicht schwierig ist für grosse Programme -> vermisst Syntax Highlighting bei Assembly Code
es sei eher zum Lernen, statt grosse Programme zu debuggen
sagt, dass man eigene Instruktion sehen kann was sie machen -> besser als Debug Outputs
-> visuelle Darstellung, was sehr hilfreich ist -> superspannend zu sehen
grosse Bereicherung
sagt es wäre gut im Quellcode zu Wissen welche Instruktion die aktuelle ist
wäre schön, wenn man Schritt zurück gehen könnte, um Animation nochmals sehen zu können
bei jump ist er verwirrt (weil Animation nicht spezifisch implementiert ist)
stellt sich vor, dass im Change Log sieht, was sich am aktuellen Zustand geändert hat.
fragt sich warum Register Scroll Balken hat, wenn man nicht scrollen kann
benutzt Arch Linux
sagt am Anfang sei es sehr viel (und unerwartet) aber man findet sich sehr schnell zurecht
findet grosse Grösse gut
interpretiert, dass CPU Memory liest und schreibt
Aufwändig sich bis zur Instruktion, die man sehen möchte durchzuklicken.
Findet Fehlerangabe im Editor schön und dass man Ctrl-Z machen kann

Nachinterview

- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
    - lädt Instruction aus dem Memory
    - und werden dann in der CPU decodiert
    - führt Instruktion aus
    - dann wird Instruction Pointer um Länge der Instruktion erhöht
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
    - Bereits richtig beschrieben, weiss dass [ ] für Adresse steht
- Was ist ein Register
    - Zwischenspeicher im Prozessor
    - kann nur mit diesen Daten arbeiten
- Was für Daten befinden sich im Register RSP
    - ist Stack Pointer zeigt auf das oberste Element auf dem Stack
    - bei pop rax: liest von Stack und schreibt in Rax und reduziert Stack Pointer um diese Grösse -> ist sich nicht sicher und geht sofort zum Editor und zum Stack Programm -> Führt aus und beantwortet nachher richtig.
- Für was stehen die verschiedenen Farben in Memory

    - gelb sind Adressen
    - ausgegraut -> nicht gebraucht, CPU hat nichts reingeschrieben
    - hell sind die benutzten
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
    - Beantwortet
- Wie hat dir der Simulator gefallen
    - hat sehr gut gefallen
    - vorallem auch push und pop -> man sieht direkt was es macht -> anstatt in der Konsole zahlen vergleichen -> sehr hilfreich visuelle Verifikation
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verstehen?
    - findet es sei extrem nützlich
    - vorallem für Personen, die keine Erfahrung haben
- Hättest du den Simulator gerne in den Übungen verwendet?
    - hätte nicht verwendet, da er sich schon gut auskennt
- Gab es Dinge die dich verwirrt haben?
- Wie findest du das design?
    - findet dunkel besser
    - findet Rundungen schön
    - findet Schrift schön
    - gefällt sehr gut
    - gut das sich Sachen unterscheiden, Cpu Cycle Box ist mehr Instrument als Teil der Cpu
    - man könnte neue Box für Memory Bus machen
- Wie findest du Abstraktion?
    - kommt sehr Nahe an was man in Bsys gelernt hat
    - entspricht dem was man in der Vorlesung lernt
- Was wünschst du dir, was fehlt noch
    - Funktionalität müsse man nicht erweitern
    - wünscht sich die Animationen abzubrechen
- Feedback zum Usability Test
    - findet gut, hat Spass gemacht
- Fragen wegen Kredits
    - Ja

Usability Test 2

Name: S23

Datum: 12. Mai

Vorinterview

- Hat beim Usability Test in der SA schon mitgemacht
- Studiengang und Semester: Informatik, 2. Semester
- Ausbildung vor Studium:
  - Informatik Lehre
  - technische Berufsmaturität
- Schätze deine Kenntnisse über Bsys1
  - ist noch vorhanden, gut
- Erkläre kurz wie der Prozessor arbeitet.
  - Prozessor hat 3 Zyklen
  - Fetch
  - Decode
  - Execute
  - Prozessor hat Pipeline und versucht diese so voll wie möglich zu halten

Test

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:
  section .text

  MOV EBX, [0x20]

  INC EAX

  ADD EAX, EBX

betrachtet Code und stellt fest, sieht aus als wird von Adresse gelesen und in ebx geschrieben

klickt auf assemble, um zu sehen was passiert

klickt Byte Size und findet es cool

klickt Play und hat das Gefühl, das alles durchläuft

fragt ob Stepping einzelne Instructions oder Cycles (Schritte des Cycles) sind

versucht die Instruktionen anzuhalten nach Play

findet verwirrend, dass bei Register nichts passiert

wollte direkt ins Memory schreiben

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - Bei add passiert nichts, weil in ebx 0 drin ist
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - Sieht Prozessorzyklus nicht
    - es werden einfach einzelne Instructions ausgeführt
- Gefällt dir das Design:
    - Was besonders
        - gefällt Register
    - Was nicht
        - Buttons sind nicht gut
        - er möchte höhere Tick Rate
        - sieht nichts zu Prozessor Zyklus

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:
  BITS 64

  mov ebx, [0x20]

  inc eax

  add eax, ebx

kann File direkt starten

erkennt unser Design vom letzten Usability Test

erzählt was er alles sieht und hovert über alle Buttons

findet Next Step Button

findet Animation sehr gut

findet move Animation auch sehr gut

klickt durch das Programm

verwirrt, ob Step fertig ist oder nicht (der Step wird nach der Ausführung immer noch Orange hervorgehoben)

vermisst, dass man schneller machen kann oder durchlaufen lassen kann

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?

    - eax und ebx werden geladen addiert, berechnet Flags und schreibt in eax und schreibt Flags
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - erklärt get instruction
    - erklärt Ausführung
    - dann Länge von instruction zum Instruction Pointer hinzugefügt
- Gefällt dir das Design:
    - Was gefällt dir besser als beim Vorherigen
        - Design viel besser, Animationen viel besser
        - CPU Zyklus Schritte klar
        - übersichtlicher
        - man merkt das Fokus auf Input und Output der Instruktionen liegt
    - Was weniger

Prozessor-Simulator 2. Runde

- Ändere nun das Programm
    - Bereits vorher ausgeführt

- Du darfst nun den Simulator frei ausprobieren

hat versucht etwas im Memory zu ändern -> würde gerne ins Memory schreiben

würde gerne automatisch durchlaufen lassen

ist zuerst verwirrt wegen Immediate Byte, dass an Current Instruction Box hängt versteht es aber nach der Animation

testet Skip Animation

möchte Jump ansehen

sagt, jump müsste Zero Flag holen

verwirrt, dass am Ende der Jump Animation nichts passiert

verwirrt, dass nach Jmp Instruction Pointer hochgezählt wird und nicht auf Wert gesetzt wird

findet cool, dass das Zeugs an den richtigen Ort fliegt

hat Stack und Base Pointer Hover entdeckt und dann von selbst Stack Programm geladen, um herauszufinden was passiert

- sagt das Zieladresse bei push auf den Stack geschrieben werden wird -> findet Animation cool
- findet Scrolling Memory cool
- sagt, dass bei pop nun Wert in rbx geschrieben wird
- Animationen machen, was er sich vorstellt

beim Programm eingeben hat er eine zu grosse Adresse eingegeben (Out of Memory) und sich gewundert, dass der Simulator einen Restart Button zeigt statt einen Fehler Dialog.

hat beim programmieren Labels eingegeben

versucht Dinge mit Labels zu programmieren

möchte sehen was passiert wenn jump Infinite Loop macht

testet “nop”

sein Infinite Loop funktioniert

- Gibt es eine Möglichkeit die Vergangenen Schritte des Programms zu sehen?
    - findet Change Log
    - würde erwarten, dass man IP Adresse sieht und was Eingabe und Ausgabe-werte waren
- Schaue nach, welchen Code du eingegeben hast.
    - findet Assembly Code Button
    - wüsste aber nicht, wo er aktuell im Assembly Code ist

Nachinterview

- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
    - Bereits richtig beantwortet
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
    - Wert innerhalb wird als Memory Adresse interpretiert und je nach Grösse vom Register, kopiert es 8 Bytes in das Register
- Was ist ein Register
    - Zwischenspeicher in der Cpu
- Was für Daten befinden sich im Register RSP
    - Stack Pointer
- Was ist das Farbige Byte im Memory
    - Bereits im Test erkannt
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
    - push rax
        - Wert von Rax wird beim Stackpointer ins Memory geladen, und Stack-pointer subtrahiert
    - pop rax
        - Stackpointer wird ausgelesen und Wert von Memory nach RAX ko-piert
- Wie hat dir der Simulator gefallen
    - Design gefällt sehr gut, farbig, nicht langweilig
    - findet cool, dass man Animationen ausschalten kann
    - und dass man Memory Transitions sieht
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verste-hen?
    - denkt schon ja
- Hättest du den Simulator gerne in den Übungen verwendet?

    - ja
- Gab es Dinge die dich verwirrt haben?
- Was wünschst du dir, was fehlt noch
    - bessere controls
    - durchlaufen lassen Button
- Abstraktion
    - findet Nahe genug an der Vorlesung
- Feedback zum Usability Test
- Fragen wegen Kredits
    - Ja gerne bei kredits dabei

hätte gerne HTML File!

Usability Test 2

Name: S24

Datum: 12. Mai 2021

Vorinterview

- Studiengang und Semester: Informatik, 2. Semester
- Ausbildung vor Studium:
  - kaufmännische Bms
- Schätze deine Kenntnisse über Bsys1
  - war auch schon besser
- Erkläre kurz wie der Prozessor arbeitet.
  - grundsätzlich Prozessor ist da, um Sachen auszurechnen mit Mmu
  - holt Daten über Data Bus und MMU
  - und lädt in seine Register

Test

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:
  section .text

  MOV EBX, [0x20]

  INC EAX

  ADD EAX, EBX

sieht Editor für Code

erwartet das Console Output gibt

schaut herum

findet Register

fragt sich was schwarze Box neben Register ist (Dezimalwert, Inhalt Register)

ist verwundert warum Buttons nicht benutzbar sind

hat Breakpoints entdeckt

sieht dass Eip erhöht

Code liest aus Speicherstelle 20 in Ebx

dann erhöht er eax

dann addiert er ebx und eax und schreibt in eax

fragt sich warum er Eip inkrementiert

aber anderes Register null bleibt

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - hat erwartet das addiert, aber hat nicht verstande warum ebx ständig 0 drinn hat
    - würde er es nicht bereits wissen, würde er es nicht verstehen
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - sagt er hat sich überhaupt nicht mit dem Memory befasst und versteht es nicht
- Gefällt dir das Design:
    - Was besonders
        - cool das es Breakpoints hat
        - Breakpoints intuitiv
    - Was nicht
        - versteht nicht was memory Teil soll

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:
  BITS 64

  mov ebx, [0x20]

  inc eax

  add eax, ebx

startet File direkt per Doppelklick

schaut sich Beispielprogramme an

startet Programm

klickt auf Assembly Code Button

führt erster Schritt aus

erklärt, dass die erste Instruktion geladen wird

versteht jetzt warum vorher alles null war (wegen used und unused bytes)

hat Instruction Pointer inkrementiert

ihm fehlt Ebx Register (da es unten nicht bei Register angezeigt wird)

nun sieht er das Rax inkrementiert wird

sagt, dass bei add zusammengerechnet wird und setzt Flags

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - holt instruktion mit Instruction Pointer, und ladet in CPU
    - sagt dass das Eax überschrieben wird
    - hat Flags vorhin erwähnt
    - IP nicht erwähnt
- Gefällt dir das Design:
    - Was gefällt dir besser als beim Vorherigen
        - viel intuitiver
        - versteht wie Memory geholt wird
        - findet die Cpu Cycle Schritte gut
        - ist verständlicher
    - Was weniger

Prozessor-Simulator 2. Runde

- Ändere nun das Programm

hat direkt gefunden

- Du darfst nun den Simulator frei ausprobieren

versteht immer noch nicht wo EBX ist

versucht einfache Addition zu machen

bekommt Fehler und versteht, was er ändern muss

immer mehr verwirrt wegen Register

fragt ob Rbx gleiche ist wie Ebx

hat Stack und Base Pointer Register Hover gesehen

konnte 20 ins Register kopieren

hat gemerkt, dass er selbst etwas überschrieben hat (bei Instruktionen mit ebx und rbx)

hat gesehn, dass neues Register dazu gekommen ist

hat etwas in den rsp geschrieben

Kann sich nicht mehr erinnern, wie Flags funktionieren

versucht Overflow Flag zu setzen in dem er grosse Zahl eingibt

schreibt zu grosse Zahl in Register EBX

hat Instruction Pointer Hover entdeckt

versucht ein Flag zu ändern mit Overflow

Schaltet kurz eine Animation aus

- Führe nun das Beispielprogramm “Stack” aus
    - jetzt sieht man, dass er Stack einfach irgendwo im Memory hat
    - sagt dass Stack nach dem zweiten push, Elemente vorne ran tut

    - man sieht schön, dass move etwas aus Memory geholt hat
    - hat Stack und Base Pointer Hover im Memory entdeckt
    - hat pop verstanden
    - versteht nicht was Wert von RSP ist -> versteht nach dem hover
    - hat push und pop verstanden
- Gibt es eine Möglichkeit die Vergangenen Schritte des Programms zu sehen?
    - findet Change Log: sieht, dass push an diese Memory Adresse drauf gepusht hat.
    - Weiss nicht was RSP ist und überlegt sich, ob das die codierte Instruktion ist (hat hier hover noch nicht entdeckt, erst später im Stackprogramm)
    - sieht, dass nach push Adresse um 8 Byte zurück geht

Nachinterview

- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
    - Prozessor lädt Instruktion vom Ort des Instruction Pointer
    - übersetzt Code und macht das
    - schreibt Daten
    - am Schluss erhöht er Instruction Pointer
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
    - sagt richtig
- Was ist ein Register
    - kleiner Zwischenspeicher im Prozessor
- Was für Daten befinden sich im Register RSP
    - Stack Pointer
- Was ist das Farbige Byte im Memory
    - Bereits erkannt
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
    - was er gesehen hat
    - Rsp raus lesen
    - sagt, dass beim push irgendwas ins Memory vorne hingetan wird
- Wie hat dir der Simulator gefallen
    - mega gute Übersicht
    - gut Cpu Cycle
    - Speed langsamer machen bei Execution Step
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verstehen?
    - Ja sehr
    - vor allem da es beim Assembly das print f nicht gibt
- Hättest du den Simulator gerne in den Übungen verwendet?
    - ja, er hat etwas gesucht, wo man sieht was in den Registern steht
- Gab es Dinge die dich verwirrt haben?
    - möchte Speed langsamer machen
- Was wünschst du dir, was fehlt noch

    - Flags Hover (Namen anzeigen)
- Feedback zum Usability Test
- Fragen wegen Kredits
    - Ja ist gut

hätte gerne html File

Usability Test 2

Name: S25

Datum: 13. Mai 2021

Vorinterview

- Studiengang und Semester: Informatik, 2. Semester
- Ausbildung vor Studium:
  - Keine Informatik Lehre
  - technische BM
- Schätze deine Kenntnisse über Bsys1
  - war sehr gut, aber ist sich nicht mehr sicher, was er noch alles weiss
- Erkläre kurz wie der Prozessor arbeitet.
  - Prozessor hat Register beispielsweise Instruction Pointer
  - dann legt er Adresse auf Speicherbus und holt Wert
  - wird beispielsweise in einem Register addiert und schreibt Wert wieder irgendwo hin

Test

Assembly debugger (x86)

- https://kobzol.github.io/davis/
- Gehe auf den Link und versuche herauszufinden, was das folgende Programm macht:
  section .text

  MOV EBX, [0x20]

  INC EAX

  ADD EAX, EBX

klickt auf auf assemble

sagt Section sei Label

findet Register mit Wert

erwartet, dass man Wert bei Slider verkleinern muss, um langsamer zu machen, merkt dass Maximum das langsamste ist und dass er es nicht langsamer machen kann

sagt hat vergessen wie add Instruktion genau funktioniert

überlegt sich ganz lange was add macht

-> sagt aber korrekt das Resultat in eax gespeichert wird -> ist sich aber nicht sicher

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - sagt Register eax und ebx werden addiert und in eax gespeichert
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - erklärt nochmals Instructions
    - sagt korrekt das Adresse 20 genutzt wird
    - lädt program neu
    - sagt, dass [20] wohl null ist findet es aber nicht
- Gefällt dir das Design:
    - Was besonders
        - Slider, aber man sollte ihn auch langsamer einstellen können
        - sagt es macht Sinn was die Register darstellen sollen wenn man es weiss
    - Was nicht
        - sagt Memory macht Sinn aber wünscht sich, dass das Memory angeschrieben ist (Adressen)
        - Knöpfe sind am Anfang verwirrend

Prozessor-Simulator 1. Runde

- Öffne das File und und versuche herauszufinden, was das folgende Programm macht:
  BITS 64

  mov ebx, [0x20]

  inc eax

  add eax, ebx

öffnet Simulator direkt ohne Probleme

klickt direkt auf Start

findet erster Eindruck cool

findet Next Step nach 3 Sekunden

findet alles selbsterklärend

findet Cpu mit Registern

sagt, dass man sieht wie Instruction geholt wird

sagt, dass mov aus dem Memory holt und in Rbx speichert

sagt, er hat vergessen, dass der Instruction Pointer noch erhöht wird

sagt oft cool

findet es sehr cool und sagt er hat Freude

sagt: hat alles Sinn gemacht dass man durch steppen kann und Schritte sieht

fragt sich, ob beim aktivierten skip Schritt übersprungen wird aber nach dem testen versteht er gleich was passiert

sagt man sieht nun Memory Adresse und sieht wie Daten geholt werden

- Was passiert bei einer Add Instruktion und welche Komponenten sind involviert?
    - sagt nimmt zwei Argumente und speichert es in erstes argument, also Register
- Kannst du uns den Ablauf des Prozessor-Zyklus bzw. die Arbeitsweise, des Prozessors anhand dem gerade eben gesehenen erklären?
    - zuerst holt Instruction
    - Instruktion ausgeführt
    - und IP erhöhen
    - sagt, dass z.B. Flags gesetzt werden oder Register oder Memory geschrieben werden
- Gefällt dir das Design:
    - Was gefällt dir besser als beim Vorherigen
        - findet auf ersten Blick visuell einfacher zu erkennen was zusammen gehört
        - Animationen helfen sehr
        - findet Controls besser, Next Step ist besser, weil es nur einen Schritt vorwärts geht
    - Was weniger
        - sagt, dass es vielleicht bei längeren Programmen zu langsam ist -> aber zum lernen sei es besser

Prozessor-Simulator 2. Runde

- Ändere nun das Programm
  geht direkt zum Editor zurück
- Du darfst nun den Simulator frei ausprobieren

geht zurück zum Editor

googelt System Calls

sucht Linux Print System Call

googelt Helloworld x86 Programm

sagt command + a geht nicht richtig im Editor, zum Teil wird die ganze Seite ausgewählt, statt nur das Programm

startet sein helloworld Programm

deaktiviert Increment Instruction Pointer

gibt Fehler im Editor ein und sieht Fehlermeldung

Führt Programm aus

Fehler bei einem Interrupt welchen wir nicht unterstützen startet Simulator neu -> Unhandled CPU Exception

Er bemerkt, dass es einen Fehler gegeben hat

sagt, dass Leute vielleicht System Calls ausführen wollen für Hello World

will Infinite Loop machen und googlet es (statt in Beispielprogrammen nach zusehen)

Google Labels

schreibt ein Programm

deaktiviert Increment Instruction Pointer

sieht, dass sein jmp Program ein Infinite Loop macht -> sagt cool

sagt, er hat sein Ziel erreicht

fragt sich, warum Immediate dort hängt und ob es einen Bug sei

sagt, es sieht absichtlich aus

- Führe nun das Beispielprogramm “Stack” aus
  - findet es direkt
  - klickt auf Assembly Code Button
  - klickt sich durch die Steps
  - sagt er hätte das wirklich gebraucht als er für die Prüfung gelernt hat!
- Gibt es eine Möglichkeit die Vergangenen Schritte des Programms zu sehen?
  - klickt direkt auf Change Log
  - sagt relevante Register und Adressen und Instructions
  - ist sich nicht ganz sicher, was drin stehen sollte, sagt bei mov würde man Argumente sehen
  - sagt bei inc vielleicht vorher und nachher
  - wüsste nicht, was er erwartet

Nachinterview

- um wieviel wird Instruction Pointer erhöht
  - sagt, dass es abhängt von der Instruction Länge
- Beschreibe den Ablauf des Prozessorzyklus. Was passiert in den Schritten?
  - holt Instruction und evaluieren diese
  - führen diese aus, je nach dem arbeiten wir mit Registern und Memory und setzten Flags
  - dann wir Instruction Pointer erhöht
- Beschreibe schrittweise was der Prozessor bei der folgenden Assembly Instruktion macht: mov rax, [0x1000]
  - sagt gleich, dass es sich um eine Adresse handelt
- Was ist ein Register
  - Bereits richtig beantwortet
- Was für Daten befinden sich im Register RSP

    - sagt er glaubt es sei der Stack Pointer und zeigt auf den aktuellen Top von Stack
    - sagt Base Pointer sei für Funktions ausführungen mit argumenten
    - sagt Base Pointer ist in rbp
- Was ist das Farbige Byte im Memory
    - Bereits erkannt
- Wie funktioniert der Stack, was passiert bei Push rax, was bei pop rax
    - mit push auf dem Stack
    - mit pop holen aber nicht löschen, sagt, dass man das sehr schön sieht
    - pop rax schiebt Wert wo Stack Pointer zeigt in Rax und um eins dekrementiert (zeigt es visuell im Simulator, eins = 1 Eintrag auf Stack, 8 Bytes)
    - zeigt push auch visuell im Simulator
    - sagt, dass bei push die Stack Pointer Adresse kleiner wird
    - sagt, dass Grösse von pop abhängt von Zielregister -> sagt aber das bei einer Zahl das Padding auf Grösse von Rax erhöht wird
    - beim testen versteht er nun den Change Log
- Wie hat dir der Simulator gefallen
    - ja, gefällt mir
    - findet gut dass man Animationen ausblenden kann
    - sagt Simulator löst das Problem, dass man im Assembly keine Print Lines machen kann
    - sagt, es vereinfacht sehr viel dass man keine Dumps mehr machen muss
    - findet es sehr gut
    - sagt UX ist sehr angenehm
    - und visuell sehr gut
- Denkst du das Programm hilft den Bsys1 Studierenden den Stoff besser zu verstehen?
    - Ja
- Hättest du den Simulator gerne in den Übungen verwendet?
    - Ja
- Gab es Dinge die dich verwirrt haben?
- Was wünschst du dir, was fehlt noch
    - bei längeren Programmen wünscht er sich ganze Instruction durch zu klicken zum schneller durchgehen
    - wünscht sich Breakpoints oder eben schneller ausführen
- Feedback zum Usability Test
- Fragen wegen Kredits
    - ja falls er nicht der einzige ist

Anhang N Protokoll Systemtest

Durchführung 14 - 11.06.2021

Test auf Branch “master” mit Pipeline-Artefakt “index.html”. Getestet mit Betriebssystem Windows 10 im Browser Google Chrome.

Datum: 11.06.2021

Tester: Eliane Schmidli

Bekannte Einschränkungen

- Bei der Ausführung als Datei werden im Web-Inspector Fehler angezeigt bezüglich dem laden von lokalen Ressourcen: “Not allowed to load local resource: file:///favicon.ico” und “Not allowed to load local resource: file://...*.js.map”. Diese zählen nicht als Fehlermeldung für diesen Systemtest.
- Tooltips und Code im Editor skalieren nicht und bleiben immer gleich gross.

Protokoll

ST-01 - Code Editor

ST-01-01 - Code Editor Starten

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
	Buttons für 4 Beispielprogramme werden angezeigt		OK
	Code für Beispiel Add wird im Textfeld angezeigt		OK

	Button Start Program wird angezeigt		OK
	Button Licenses rechts unten wird angezeigt		OK
Code Beispiel Buttons drücken	Wenn man auf die Buttons klickt werden die Beispielprogramme im Textfeld ersetzt		OK

ST-01-02 - Code Editor Programmieren

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Zeile im Programm im Textfeld zur Hälfte löschen	Code sollte bearbeitbar sein		OK
Start Program Klicken	Fehlermeldung erscheint mit Zeilenangabe		OK
	Der Start Program Button zeigt nach dem erscheinen der Meldung keinen Loading Spinner mehr		OK
Kompletten Code bis auf “BITS 64” löschen	Textfeld zeigt nur “BITS 64” an		OK
Start Program Klicken	Fehlermeldung, dass Code leer ist erscheint.		OK
	Der Start Program Button zeigt nach dem erscheinen der Meldung keinen Loading Spinner an		OK

Beispielprogramm “add” auswählen	Code wird angezeigt		OK
Start Program klicken	Fehlermeldung verschwindet		OK
	Loading Spinner erscheint		OK
	Navigiert zu Simulator		OK

ST-01-03 - Code Editor Navigation

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Beispielprogramm Stack auswählen	Programm Stack wird in Textfeld geladen		OK
Start Program klicken	Loading Spinner auf Button erscheint		OK
	Navigation zu Simulator		OK
Back To Editor klicken	Loading Spinner erscheint		OK
	Navigation zu Editor		OK
	Editor zeigt Programm Stack in Textfeld an		OK
Page reloaden	Navigation zu Editor		OK
	Editor zeigt Programm Stack in Textfeld an		OK

ST-01-04 - Lizenzen anzeigen

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**

Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Button Licenses unten rechts klicken	Menu öffnet sich		OK
	Titel: “Licenses for GCSx64 - x86_64 Graphical CPU Simulator” wird angezeigt		OK
	Unsere Lizenz mit Titel: “GCSx64 - x86_64 Graphical CPU Simulator” Wird als erstes angezeigt		OK
	Die Lizenz wird schön dargestellt (mit Zeilenumbrüchen)		
Nach unten scrollen bis zu leerer Tabellen Zeile	Verschiedene Lizenzen werden aufgelistet		OK
	Unter der leeren Zeile wird der gesamte Text der GPL eingeblendet “SPDX-License-Identifier: GPL-2.0-only”		OK
	Unterhalb dieser Lizenz befindet sich kein weiterer Eintrag		OK
Neben das Menu klicken	Das Menu schliesst sich		OK

ST-02 - Simulator

ST-02-01 - Simulator anzeigen

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt (mit add Programm)		OK
Start Program klicken	Navigation zu Simulator		OK
	3 CPU Steps mit Toggle auf Haken werden angezeigt.		OK
	Bei Hover über Toggles wird Text angezeigt.		OK
	Back to Editor und Next Step Button werden angezeigt		OK
	Bei Hover über Button wird Text angezeigt.		OK
	Step: 1 Get Instruction ist anders eingefärbt als die anderen Schritte		OK
	CPU Komponente mit Titel “CPU” und Komponenten Current Instruction, Instruction Pointer und Registern wird angezeigt CPU-Box (grün) wird leer angezeigt.		OK
	Current Instruction hat Titel “Current Instruction”, zeigt 8 Bytes mit Inhalt “00” an, und ein leeres Instruktionsfeld.		OK

	Instruction Pointer hat Titel “Instruction Pointer” und ein vierstelliges Adressfeld auf “0000”		OK
	Register hat Titel “Register” und 2 Register mit Namen “RAX” und “RBX” und jeweils 8 Bytes mit Inhalt “00” Zusätzlich 2 Register “RBP” und “RSP” mit 8 Bytes wobei das zweite “03” und der Rest “00” anzeigt.		OK
	Register RSP und RBP zeigen eine Adresse auf 0300		OK
	Bei Hover über Register RSP, RBP und Adressen wird Text Stack bzw. Base Pointer Register angezeigt.		OK
	Flags werden angezeigt: CF, OF, ZF, SF mit Inhalt “0”		OK
	Memory hat Titel “Memory” und Memorydatenbereich		OK
	Memorydaten haben oben Adresszeile von +1 bis +F. Jede Datenzeile beinhaltet 4 Stellige Hex Adresse und Bytes.		OK
	Instruction Code wird als ausgefüllte Bytes dargestellt. Darauffolgende Bytes sind leicht ausgeblendet. (Ausser Stack-Pointer Byte)		OK
Hover über erstes Byte	Erstes Byte wird umrandet und hat einen leichten gelbstich.		OK

	Bei Hover erscheint Instruction Pointer Beschriftung.		
Scrolle Memory Daten	Scrollen von 0000 bis zur Adresse 0FF0 ist möglich. Alle Zeilen sind mit Bytes gefüllt		OK
Scrolle zu 0300	An Stelle 0300 wird ein Farbiges Byte dargestellt. Oben und Unten befindet sich ein Pfeil		OK
Hover über Farbiges Byte	Bei Hover über Byte erscheint Stack und Base-Pointer Beschriftung. Stack links und Base unterhalb des Bytes.		OK
Hover über Assembly Code und Change Log Button	Rechts werden zwei Buttons Assembly Code und Change Log angezeigt		OK
	Bei Hover werden Texte angezeigt.		OK
Klicke auf Assembly Code	Assembly Code zeigt beim Klicken den Code das Default Programm Move and Add an		OK
Klicke auf Change Log	Change Log zeigt “Nothing has Changed yet” an.		OK

ST-02-02 - Durchlauf zwei Instruktionen und Restart

Hinweis: Falls die Animationen zu schnell sind, kann die Programmausführung beliebig oft wiederholt werden, in dem man einen Refresh im Browser macht.

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib Program ein:
BITS 64
mov rax, [0x0]
mov rbx, [0x30]
	Eingabe möglich		OK

Start Program klicken	Navigation zu Simulator		OK
	Get Instruction ist angewählt		OK
	Alle Toggles sind auf Haken gesetzt.		OK
	Im Memory sind nur 16 Bytes hervorgehoben		OK
Step Button drücken	Pulse auf Current Instruction erscheint 8 Bytes (Adresse 0000) im Memory werden gross und schräg und bewegen sich nachher zu Current Instruction und ersetzen die Bytes dort. Pulse auf Assembly das erscheint.		OK
	Assembly Übersetzung zeigt mov rax, [0x0]		OK
	CPU-Box (Grün) zeigt mov in der Mitte. Links steht [0x000] und eine leere Graue Zeile erscheint auf derselben Höhe Rechts steht RAX		OK
	Step zeigt Get Instruction an.		OK

	Die 8 Bytes im Memory sind nun alle leicht gelb. Nur das erste wird Blau umrandet.		OK
	Alle anderen Daten bleiben unverändert.		OK
Klicke auf Change Log	Change Log zeigt: Nothing has changed yet		OK
	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK
Step Button drücken	Pulse Assembly; Erste 8 Bytes von Memory Daten werden gross und schräg und bewegen sich zu linker Zeile von CPU-Box und bleiben dort liegen. CPU-Box fährt nach links. Rechte Zeile auf Höhe Beschriftung RAX tritt zum Vorschein und zeigt Bytes: 48 8B 04 25 00 00 00 00 Pulse auf RAX Register Bytes bei Box werden geklont und fliegen zu RAX Register und bleiben dort liegen. CPU-Box fährt nach rechts. Linke Zeile ist verschwunden, Beschriftungen der Zeilen an		OK

	CPU-Box verschwinden (zeigt nur mov an)		
	Step hat sich bei Beginn der Animation auf “2 Execute Instruction” geändert		OK
	Alle anderen Daten bleiben unverändert		OK
Klicke auf Change Log	Change Log zeigt: mov rax, [0x0] RAX: 48 8B 04 25 00 00 00 00 Box mit 1		OK
	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK
Step Button drücken	Pulse Instruction Pointer, Dann grosse, schräge Current Instruction Bytes; Pulse auf Instruction Pointer Adresse Current Instruction Bytes fliegen zu Instruction Pointer und legen sich übereinander. Der Instruction Pointer ändert auf “0008”		OK
	Step hat sich zu Beginn der Animation auf “3 Increment Instruction Pointer” geändert.		OK
	In Memory wird nun Byte an 0008 Blau umkreist.		OK

	Die Bytes zuvor sind nicht mehr gelb und Byte an 0000 nicht mehr umrandet.		
	Alle anderen Daten bleiben unverändert.		OK
Klicke auf Change Log	Change Log zeigt: mov rax, [0x0] RAX: 48 8B 04 25 00 00 00 00 Box mit 1		OK
	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK
Step Button drücken	Step Button ist während der Animation disabled.		OK
	Es werden analog zum ersten Durchlauf die Bytes von 0008 bis 000F geklont und zu Current Instruction gebracht.		OK
	Assembly Übersetzung “mov rbx, [0x30]” wird angezeigt		OK
	CPU-Box zeigt mov und links Zeile und [0x0030], rechts RBX		OK
	Step ändert auf “1 Get Instruction”		OK
	Bytes von 0008-000F haben Gelbstich, Stelle 0008 ist umrandet.		OK
	Alle anderen Daten bleiben unverändert.		OK

	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK
Step Button drücken	Step Button ist während der Animation disabled.		OK
	Bytes von 0030 bis 0037 werden eingeblendet Es werden analog zum ersten Durchlauf die Bytes von 0030 bis 0037 geklont und zu linken Zeile der Box gebracht. Die Box bewegt sich nach links, Zeile mit 8 Mal 00 erscheint und fliegt zu RBX.		OK
	Register “RBX” zeigt “00 00 00 00 00 00 00 00” an		OK
	Register “RAX” zeigt “48 8B 04 25 00 00 00 00” an		OK
	Step ändert auf “2 Execute Instruction”		OK

Klicke auf Change Log	Change Log zeigt:
	mov rax, [0x0]
	RAX: 48 8B 04 25 00 00 00 00
	Box mit 1
	mov rbx, [0x30]
	RBX: 00 00 00 00 00 00 00 00
	Box mit 2
			OK

	Alle anderen Daten bleiben unverändert.		OK
	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK

Step Button drücken	Step Button ist während Animation disabled		OK
	Current Instruction Bytes fliegen zu Instruction Pointer und legen sich übereinander. Der Instruction Pointer ändert auf “0010”		OK
	Byte an Stelle 0010 ist ausgeblendet, hat aber Rahmen. Rahmen bei 0008 verschwunden		OK
	Button Next Step zeigt nun Restart an. Bei Hover erscheint Text mit restart		OK
	Steps sind alle grau		OK
	Alle anderen Daten bleiben unverändert.		OK
	In der Konsole des Inspectors wird kein Fehler angezeigt.		OK
Restart drücken	Seite wird neu geladen		OK
	Stand wie zu Beginn wird angezeigt: Get Instruction hervorgehoben Button Next Step (nicht Restart) Current Instruction Leer mit 00 Bytes CPU-Box Leer Instruction Pointer 000 Register RAX, RBX 0 Erstes Byte im Memory umrandet		OK

Klicke auf Change Log	Change Log zeigt: Nothing has changed yet		OK

ST-02-03 - Reload Simulator

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Step Button drücken	Get Instruction wird ausgeführt		OK
Seite Refreshen	Seite wird neu geladen		OK
	Status quo wird hergestellt, Step = Get Instruction usw.		OK

ST-03 - Skip Animations

ST-03-01 - Skip One Animation

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Erster Toggle Get Instruction ausschalten	Toggle zeigt X und Farbe verschwindet		OK
	Hover Text sagt Animations for this step are disabled		OK

	Andere Toggles inkl. Hover Text bleiben unverändert		OK
Step Button drücken	Get Instruction wird nicht animiert		OK
	Current Instruction zeigt Assembly Code und Instruction Bytes an		OK
	CPU-Box zeigt [0x000], mov und RAX Links erscheint Zeile		OK
	Memory sind ersten 8 Bytes leicht gelb und erstes Byte umrandet.		OK
Step Button drücken	Execution Instruction wird animiert		OK
Step Button drücken	Increment Instruction wird animiert.		OK
Erster Toggle einschalten	Hover Text: Animation for this step are enabled		OK
	Haken und Farbe erscheinen		OK
Dritter Toggle ausschalten	Analog zu vorher wird Toggle und Text geändert. Andere Toggles bleiben unverändert		OK
Step Button drücken	Get Instruction wird animiert		OK
Step Button drücken	Execution Instruction wird animiert		OK
Step Button drücken	Increment Instruction Pointer wird nicht animiert		OK
	Instruction Pointer zeigt 0010		OK

	Memory Byte 0010 wird umrahmt, 0008 nicht mehr		OK
Dritter Toggle einschalten	Hover Text: Animations for this step are enabled		OK
	Haken und Farbe erscheinen		OK
Zweiter Toggle ausschalten	Analog zu vorher wird Toggle und Text geändert. Andere Toggles bleiben unverändert		OK
Step Button drücken	Get Instruction wird animiert		OK
Step Button drücken	Execute Instruction wird nicht animiert		OK
	Step Execute Instruction ist hervorgehoben		OK
	CPU-Box zeigt nur noch add		OK
	RBX zeigt Werte von RAX an		OK

Change Log Klicken	zeigt:
	mov rax, [0x0]
	RAX: 48 8B 04 25 00 00 00 00
	Kasten mit 1
	
	mov rbx, [0x30]
	RBX: 00 00 00 00 00 00 00 00
	Kasten mit 2
	
	add rbx, rax
	RBX: 48 8B 04 25 00 00 00 00
	Flags: CF: 0, SF: 0, ZF: 0, OF: 0
			OK

	Kasten mit 3		
Alle Toggles einschalten und mit Step Button durch Programm gehen	Alle Schritte werden animiert.		OK

ST-04 - Diverse Instruktionen

ST-04-01 - Programm mit allen Access Möglichkeiten

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib Program ein:
BITS 64
mov rax, [0x0]
mov [0x80], eax
add ax, cx
cmovnl rax, rbx
xor rbx, rbx
jz hello
hello:
add [0x60], rax
mov rbx, qword [rbp + 0x10]
mov rax, qword [rbp*2]
sub r8w, 0x30
	Eingabe möglich		OK

Start Program klicken	Navigation zu Simulator		OK

Toggle 1 und 3 aus-schalten	Execute Instruction ist einziger der noch eingeblendet ist.		OK
Next Step klicken	mov rax, [0x0] wird angezeigt		OK
	CPU-Box zeigt [0x0000] als Input und RAX als Output		OK
Next Step klicken	8 erste Bytes von Memory werden gelesen, CPU-Box geht nach links 8 Bytes fliegen zu RAX		OK
	RAX zeigt 48 8B 04 25 00 00 00 00		OK
Next Step 2-mal klicken	Mov[0x80], eax wird angezeigt		OK
	CPU-Box zeigt EAX als Input und [0x0080] als Output		OK
	Bytes in Memory an 0x0080 sind ausgeblendet		OK
Next Step klicken	4 erste Bytes von RAX werden gelesen, CPU-Box geht nach links An Stelle 0080 werden 4 Bytes eingeblendet 4 Bytes von CPU-Box zu Stelle 0080		OK
	Register RCX wird nicht ange-zeigt		OK
Next Step 2-mal klicken	add ax, cx wird angezeigt		OK
	CPU-Box zeigt AX und CX als Input und als Output AX und Flags		OK

	RCX wird bei den Registern angezeigt		OK
Next Step klicken	2 erste Bytes von RAX und RCX werden nacheinander gelesen, CPU-Box geht nach links 2 Bytes werden in erste 2 Bytes von RAX geschrieben 4 Flags werden geschrieben, SF ist 1		OK
	AX zeigt 48 8B		OK
Next Step 2-mal klicken	cmovnl rax, rbx wird angezeigt		OK
	CPU-Box zeigt RAX, RBX und Flags als Input und als Output RAX		OK
Next Step klicken	Flags werden parallel gelesen, Dann RAX und dann RBX CPU-Box geht nach links RAX wird geschrieben		OK
Next Step 2-mal klicken	xor rbx, rbx wird angezeigt		OK
	CPU-Box zeigt RBX und RBX als Input und als Output RBX und Flags		OK
Next Step klicken	RBX wird zwei Mal nacheinander gelesen und in verschiedenen Zeilen abgelegt CPU-Box bewegt sich nach links Flags werden parallel geschrieben, SF wird 0, ZF 1		OK
Next Step 2-mal klicken	je 0x1b wird angezeigt		OK

	Immediate 1B wird in Current Instruction angezeigt		OK
	Zeilen neben CPU-Box sind leer (keine Flags)		OK
	CPU-Box zeigt Immediate und Flags als Input und als Output nichts		OK
Next Step klicken	Flag ZF wird gelesen und die Immediate bewegt sich von der Instruction Box zum Input. Die CPU-Box bewegt sich nach links und nach rechts. Die Immediate ist verschwunden.		OK
Next Step 2-mal klicken	add [0x60], rax wird angezeigt		OK
	CPU-Box zeigt [0x0060] und RAX als Input und als Output [0x0060] und Flags		OK
	Bytes im Memory an Adresse [0x0060] sind ausgegraut		OK
Next Step klicken	8 Bytes an Stelle 0x0060 werden eingeblendet. 8 Bytes aus RAX bewegen sich zu Input. 8 Bytes aus Memory bewegen sich zum Input. CPU-Box fährt nach links. 8 Bytes werden ins Memory an 0x0060 geschrieben Alle Flags werden parallel geschrieben		OK

	0x0060 zeigt selber Wert wie RAX und 8 Bytes sind eingeblendet		OK
	Flag ZF zeigt 0		OK
Next Step 2-mal klicken	mov rbx, [rbp+0x10] wird angezeigt		OK
	CPU-Box zeigt [0x0310] und RBP als Input und als Output RBX		OK
Next Step klicken	8 Bytes aus RBP bewegen sich zu Input. 8 Bytes aus Memory [0x0310] bewegen sich zum Input. CPU-Box fährt nach links. 8 Bytes werden in Register RBX geschrieben		OK
	RBX zeigt 8 mal 00		OK
Next Step 2-mal klicken	mov rax, [rbp+rbp+0x0]		OK
	CPU-Box zeigt [0x0600] und RBP als Input und als Output RAX		OK
Next Step klicken	8 Bytes aus RBP bewegen sich zu Input. 8 Bytes aus Memory [0x0600] bewegen sich zum Input. CPU-Box fährt nach links. 8 Bytes werden in Register RAX geschrieben		OK
	RAX zeigt 8 mal 00		OK
Next Step 2-mal klicken	sub r8w, byte +0x30 wird angezeigt		OK

	Register R8 wird angezeigt		OK
	Immediate 30 wird angezeigt		
	CPU-Box zeigt Immediate und R8W als Input und als Output R8W und Flags		OK
Next Step klicken	Zwei Bytes von R8 werden gelesen, Dann die Immediate danach fährt CPU nach links, 2 Bytes werden in r8 geschrieben und 4 Flags werden parallel geschrieben		OK
	CF und SF sind 1		OK
	R8 zeigt D0 FF und 6 mal 00		OK

Changelog zeigt	mov rax, [0x0]
	RAX: 48 8B 04 25 00 00 00 00
	1
	mov [0x80], eax
	[0x0080]: 48 8B 04 25
	2
	add ax, cx
	AX: 48 8B
	Flags: CF: 0, OF: 0, ZF: 0, SF:
	1
	3
	cmovnl rax, rbx
	RAX: 48 8B 04 25 00 00 00 00
	4
	xor rbx, rbx
	RBX: 00 00 00 00 00 00 00 00
	Flags:CF: 0, OF: 0, ZF: 1, SF:
	0
	5
			OK

	je 0x1b
	6
	add [0x60], rax
	[0x0060]: 48 8B 04 25 00 00 00 00
	Flags: CF: 0, OF: 0, ZF: 0, SF: 0
	7
	mov rbx, [rbp+0x10]
	RBX: 00 00 00 00 00 00 00 08
	mov rax, [rbp+rbp+0x0]
	RAX: 00 00 00 00 00 00 00 09
	sub r8w, byte +0x30
	R8W: D0 FF
	Flags: CF: 1, OF: 0, ZF: 0, SF: 1		

ST-04-02 - Scroll into view

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib Program ein:
BITS 64
add rcx, rdx
add r8w, r9w
add r10, r11
mov [0x0800], r11d
mov eax, [0x0800]
	Eingabe möglich		OK

push qword [0x0800] pop r11			
Start Program klicken	Navigation zu Simulator		OK
Alle Animationen ausschalten	Alle Toggles zeigen X		OK
Next step 10-mal drücken, bis mov in CPU-Box steht	Folgende Register werden angezeigt: RAX RBX RBP RSP RCX RDX R8 R9 R10 R11 Und zu unterst Flags		OK
Bei Registern nach oben scrollen			OK
Falls R11 nicht verdeckt wird, Höhe des Browserfenster verkleinern			OK
Animationen einschalten	Alle Toggles zeigen Haken		OK
Next Step klicken	Bytes von Register R11 werden gescrollt bis sie ersichtlich sind und fliegen dann zur CPU-Box		OK

	Memory wird zu Adresse 0800 gescrollt und Daten fliegen von CPU-Box zur Adresse		
Next Step klicken	Nichts wird gescrollt		OK
Next Step klicken	Memory scrollt zu Adresse 0012		OK
Next Step klicken	Memory scrollt zu 0800 und Daten fliegen zu CPU-Box. Register scrollt zu RAX und Daten fliegen von CPU-Box dahin		OK
Next Step klicken	Nichts wird gescrollt		OK
Schritt 1 und 3 Animation ausschalten	2 ist noch enabled		OK
Next Step 2-mal klicken	Zuerst werden Daten aus 0800 gelesen und beim Schreiben scrollt das Memory dann zum Stack nach 02F8 und fügt die Daten dort ein.		OK
Next Step 2-mal klicken	Nichts scrollt		OK
Next Step klicken	Beim schreiben scrollen die Register zu R11 und dann zurück zu RSP		OK

ST-04-03 - Zeilenübergreifender Memory Access

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib Program ein:
BITS 64
mov rax, [0x3d]
mov [0x4a], rax
	Eingabe möglich		OK

Start Program klicken	Navigation zu Simulator		OK
Toggle 1 und 3 ausschalten	Toggle 2 ist eingeschaltet		OK
Next step 2-mal klicken	Bytes von 003d-003f und 0040-0044 werden eingeblendet, werden schräg und gross und bewegen sich zu CPU-Box		OK
Next step 3-mal klicken	Beim Schreiben werden Bytes 004A-004F und 0050-0051 eingeblendet. Dann werden sie schräg und gross Dann fliegen die Bytes von der CPU-Box an diese Stelle		OK

ST-05 - Stack und Base-Pointer Bytes

ST-05-01 - Darstellung und Animation Stack- und Base-Pointer

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib dieses Programm ein:
BITS 64
push 0x6
push qword [0x6]
mov rbp, 0x2f8
pop rax
push 6
mov rsp, 0x100
mov rbp, 0x200
	Eingabe möglich		OK

Start Program klicken	Navigation zu Simulator		OK
	Register RSP zeigt 00 03 in den ersten zwei Bytes und Adresse 0300 RBP zeigt dasselbe an		OK
Scrolle zu 0300	Ein Byte wird farbig dargestellt. Es hat ein Pfeil oben und unten		OK
Über Stack Byte Hovern	Links wird Stack Pointer, unten Base Pointer angezeigt		OK
Hover über Register RBP	Base Pointer Register wird angezeigt		OK
Hover über Register RSP	Stack Pointer Register wird angezeigt		OK
Schritt 1 und 3 Animation ausschalten	Nur Schritt zwei ist enabled		OK
2-mal next Step klicken	Sobald Wert in RSP geschrieben wird (F8 02) wird Adresse angepasst und im Memory werden die neuen Stack Bytes Rot und das vorderste (06) erhält		OK

	den roten Pfeil, der Animiert wird mit einem Puls.		
Zu Base Pointer scrollen	Base Pointer ist alleine auf Byte 0300 und wird blau mit blauen Pfeil unten dargestellt		OK
Hover über Base Pointer	Base Pointer wird angezeigt		OK
Hover über Stack Pointer	Stack Pointer wird angezeigt.		OK
3-mal next Step klicken	Sobald RSP beschrieben wird, ändert sich die Adresse auf 02F0 und die vorderen 8 Bytes der Zeile 02F0 sind rot. Stack Pointer zeigt auf vorderstes Element. Die hinteren 8 Byte sind weiss		OK
	Der Base Pointer ist immer noch bei 0300		OK
Next Step 3-mal klicken	Sobald F8 02 in RBP geschrieben ist, passt sich die Adresse an. Die hinteren 8 Bytes von Zeile 02F0 werden nun blau angezeigt, das vorderste Byte der 8 (06) hat einen blauen Pfeil, der eine Pulse Animation macht.		OK
Hover über Base Pointer	Text Base Pointer wird angezeigt		OK
Next Step 3-mal klicken	Sobald RSP beschrieben wird, werden die hinteren 8 Byte der Zeile 02f0 rot und beide Pointer		OK

	zeigen auf Byte 06, beide Pulsieren.		
	Die vorderen 8 Bytes sind weiss und nicht ausgegraut.		OK
Next Step 3-mal klicken Sobald Memory beschrieben wird weg scrollen	Keine Pfeil Puls animation erscheint		OK
Zu Memory Zeile 0100 scrollen	Alle Bytes sind ausgegraut		OK
Next Step 3-mal klicken	Stack Byte wird mit Pointer angezeigt, Schrift ist schwarz		OK
Zu Memory Zeile 0200 scrollen	Alle Bytes sind disabled		OK
Next Step 3-mal klicken	Base Byte wird mit Pointer angezeigt, Schrift ist schwarz		OK

ST-05-02 - Darstellung Stack-, Base-Pointer- und Instruction-Bytes

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK

Gib Program ein:
BITS 64
mov rsp, 0
mov rbp, 8
add rax, rbx
add rax, rbx
	Eingabe möglich		

Start Program klicken	Navigation zu Simulator		OK

Schritt 1 und 3 Animationen ausschalten	Schritt 2 ist enabled		OK
Next Step 2-mal drücken	8 Bytes von Zeile 0000 werden rot, erstes Byte ist Blau umrandet und hat einen roten Pfeil oben.		OK
Hovern über 1. Byte	Instruction Pointer und Stack Pointer wird angezeigt, wobei “Pointer” von Stack Pointer von Instruction Pointer überdeckt wird		OK
Next Step 2-mal klicken	Bytes an Stelle 0008 und 0009 werden gelb		OK
	Byte 0005 wird blau umrandet		OK
	Byte 0000 ist rot und hat Pointer		OK
Next Step klicken	Bytes an Stelle 0008 bis 000f werden blau		OK
	Byte an Stelle 0008 hat Pointer		OK

ST-06 - Browser Navigation

ST-06-01 - Browser Navigation

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
	Adresse zeigt: .../x86/dist/index.html#/editor		OK

Editor aus Adresse löschen Adresse: .../x86/dist/index.html#/ Und öffnen	Adresse zeigt: .../x86/dist/index.html#/editor		OK
Start Program klicken	Link zeigt: ...x86/dist/index.html# /simulator /48,8B …. /QklUUyA….		OK
Teil Qk… löschen Adresse: ...x86/dist/index.html# /simulator /48,8B…./	Code Editor wird angezeigt .../x86/dist/index.html#/editor		OK
Start Program klicken	Link zeigt: ...x86/dist/index.html# /simulator /48,8B …. /QklUUyA….		OK
Teil /Qk… löschen Adresse: ...x86/dist/index.html# /simulator /48,8B….	Code Editor wird angezeigt .../x86/dist/index.html#/editor		OK
Start Program klicken	Link zeigt: ...x86/dist/index.html# /simulator /48,8B …. /QklUUyA….		OK
Teil nach simulator löschen	Code Editor wird angezeigt .../x86/dist/index.html#/editor		OK

Adresse: ...x86/dist/index.html# /simulator/			
Swap Programm auswählen	Swap Programm erscheint		OK
Start Program klicken	Link zeigt: ...x86/dist/index.html# /simulator /48,8B …. /QklUUyA...		OK
Teil 48,8B… löschen Adresse: ...x86/dist/index.html# /simulator ***//***QklUUyA...	Code Editor wird angezeigt Swap Program ist ausgewählt		OK
Start Program klicken	Link zeigt: ...x86/dist/index.html# /simulator /48,8B …. /QklUUyA...		OK
Link aus Adresszeile kopieren			OK
Neuer Tab öffnen und beliebige Seite aufrufen			OK
Link von simulator in Adresszeile eingeben	Simulator öffnet sich		OK
Assembly Code öffnen	Zeigt Swap Program		OK
Back to Editor klicken	Editor öffnet sich mit Swap Program		OK

ST-07 - Darstellung Assembly Code im Menu

ST-07-01 - Darstellung Assembly Code

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Gib Program ein: BITS 64 mov rax, [0x0] ; Add	Eingabe möglich		OK
Nach Kommentar ;aaa… Einfügen bis aaaaa… 4 Zeilen lang			OK
40 Zeilenumbrüche anhängen Zu underst ;hello einfügen			OK
Start Program klicken	Navigation zu Simulator		OK
Assembly Code öffnet	Fenster geht nicht über Bildrand heraus und man kann in 2 Richtungen scrollen		OK
	Aaaaa.. Wird auf einer Zeile dargestellt und man kann bis zum Schluss horizontal scrollen		OK

	man kann vertikal scrollen bis zu ;hello (man kann noch ein wenig weiterscrollen als ;hello)		OK

ST-08 - Responsive Design

ST-08-01 - Höhe 900

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Inspect öffnen			OK
Fenster mindestens 900 Pixel hoch machen und maximal breit			OK
Langsam Fenster Breite bis 560 Pixel verringern	Simulator sollte schön skalieren und immer wieder Padding verringern, um möglichst lange leserlich zu bleiben		OK
Fenster wieder maximal verbreitern			OK
Langsam Fenster Höhe auf 800 reduzieren	Simulator sollte schön skalieren		OK

ST-08-02 - Höhe 800

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**

Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Inspect öffnen			OK
Fenster ungefähr 800 Pixel hoch machen und maximal breit			OK
Langsam Fenster Breite bis 560 Pixel verringern	Simulator sollte zuerst zusammengeschoben werden und dann schön skalieren und immer wieder Padding verringern, um möglichst lange leserlich zu bleiben		OK
Fenster wieder maximal verbreitern			OK
Langsam Fenster Höhe auf 650 reduzieren	Simulator sollte schön skalieren		OK
			OK

ST-08-03 - Höhe 650

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Inspect öffnen			OK
Fenster ungefähr 650 Pixel hoch machen und maximal breit			OK

Langsam Fenster Breite bis 560 Pixel verringern	Simulator sollte zuerst zusammengeschoben werden und dann schön skalieren und immer wieder Padding verringern, um möglichst lange leserlich zu bleiben		OK
Fenster wieder maximal verbreitern			OK
Langsam Fenster Höhe auf 500 reduzieren	Simulator sollte schön skalieren		OK

ST-08-04 - Höhe 500

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Start Program klicken	Navigation zu Simulator		OK
Inspect öffnen			OK
Fenster ungefähr 500 Pixel hoch machen und maximal breit			OK
Langsam Fenster Breite bis 560 Pixel verringern	Simulator sollte zuerst zusammengeschoben werden und dann schön skalieren und immer wieder Padding verringern, um möglichst lange leserlich zu bleiben		OK
Next Step 3-mal klicken	Animationen werden klein aber schön dargestellt		OK

ST-08-05 - Editor

Aktionen	**Erwartetes Resultat**	**Kommentar von Tester**	**Status**
Doppel klicke Datei index.html	Code Editor wird angezeigt		OK
Fenster maximal gross machen			OK
Langsam Fensterbreite und Höhe verringern	Editor sollte skalieren		OK