OST
Ostschweizer
Fachhochschule

Weiterentwicklung Prozessor-Simulator

Studienarbeit

Studiengang Informatik

OST - Ostschweizer Fachhochschule

Campus Rapperswil-Jona

Herbstsemester 2021

Autoren: Michael Schneider, Tobias Petter

Betreuer: Prof. Stefan Richter

Inhaltsverzeichnis

Einleitung

Zu den Grundlagen der Ausbildung eines Informatikers gehört ein fundiertes Verständnis für die Funktionsweise von Computersystemen. Obwohl heutzutage zunehmend die höheren Abstraktionsebenen wie beispielsweise Betriebssysteme und Applikationen in den Vordergrund treten, ist ein Verständnis von hardwarenahen Prozessen nach wie vor unerlässlich. Zentral in jedem Hardwaresystem ist der Prozessor. Einfach gesagt nimmt dieser Instruktionen entgegen und verarbeitet sie, indem er Speicherinhalte modifiziert.

Die Funktionsweise von modernen Prozessoren wird an der OST im Rahmen des zweiteiligen Moduls "Betriebssysteme" unterrichtet. Da der Inhalt oft befremdlich auf neu angehende Informatik-Studenten wirken kann, wurde die Entwicklung eines Simulators angedacht, in dem die Arbeitsschritte eines Prozessors visuell Schritt für Schritt nachverfolgt werden können. Dieser Simulator wurde im Herbstsemester 2020 sowie im Frühjahrssemester 2021 von Eliane Schmidli und Yves Boillat im Rahmen ihrer Studien- respektive Bachelorarbeit entwickelt.

Das Ziel der vorliegenden Arbeit war, dieses Produkt weiterzuentwickeln. Dabei standen Verbesserungen der Benutzbarkeit, eine Erhöhung des Lerneffekts bei den Benutzern sowie das Implementieren neuer Features im Vordergrund. Der Simulator soll so künftigen Generationen von Studenten noch besser beim Verständnis dieses grundlegenden Stoffes helfen.

Ausgangslage

Das von den Kollegen Schmidli und Boillat bereitgestellte Produkt ist bereits sehr gut. Es beinhaltet einen Editor, in dem NASM-Assembly-Code eingegeben werden kann. Dieser Code kann dann im Simulator Schritt für Schritt ausgeführt werden. Für jede Instruktion ist dabei der Prozessorzyklus in die drei Schritte "Get Instruction", "Execute Instruction" und "Increment Instruction Pointer" aufgeteilt. Eine Infobox informiert über den aktuellen Stand im Zyklus, während zwei grosse Behälter den aktuellen Zustand von Prozessor und Speicher zeigen.

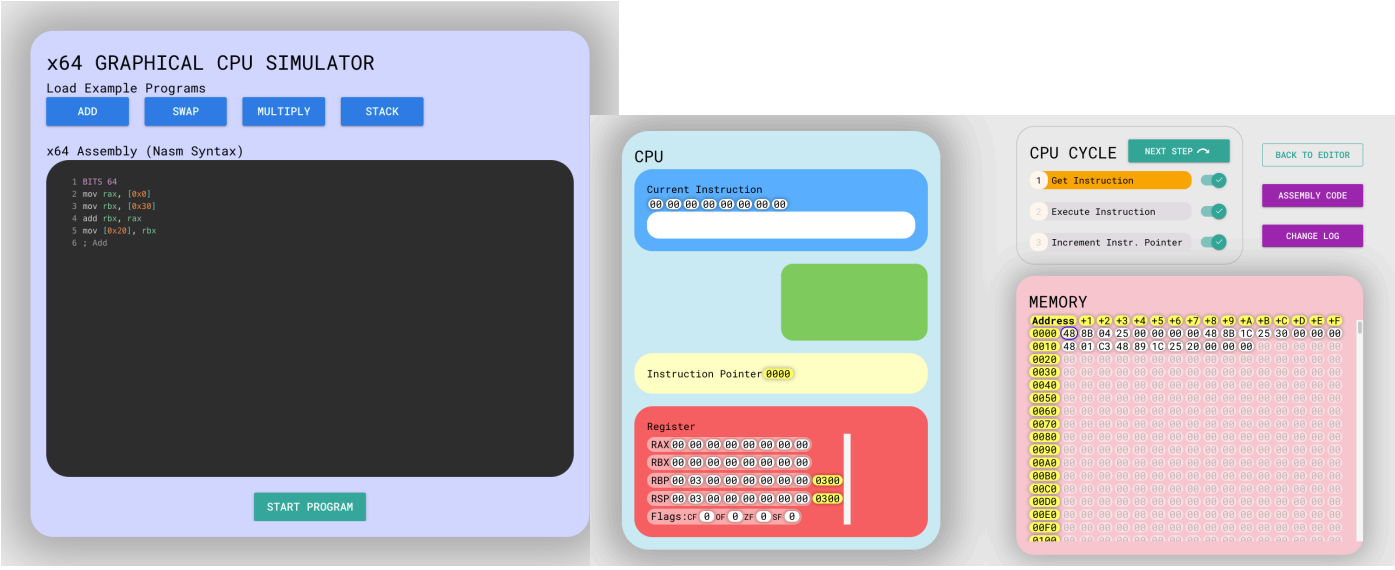

Abbildung 1 - Die Originalversion des Simulators

Sämtliche Zustandsänderungen sind animiert, um dem Benutzer klar zu machen, was von wo wohin kopiert wird. Dabei wird die Aufmerksamkeit des Benutzers durch kleine Zusatzbewegungen auf die Bereiche des Bildschirms geführt, wo bald Veränderungen stattfinden.

Ausserdem ist es möglich, sich den im Editor eingegebenen Assembly-Code auch während der Ausführung des Programmes anzusehen. Hierbei ist allerdings keine Hervorhebung der Syntax gegeben. Ausserdem informiert ein Change-Log über alle Änderungen, die sich seit Beginn des Programmes in den Registern oder dem Speicher ergeben haben.

In unseren anfänglichen Begutachtungen haben wir den bestehenden Simulator als vollwertiges und einsatzfähiges Produkt erlebt, das sich durch eine klar verständliche Struktur sowie Liebe zum Detail in Form von vielen kleinen Zusatzfunktionen auszeichnete. Dies hat sich auch durch unseren User-Test bestätigt, in dem eigentlich alle befragten Studenten den Simulator als einfach verständlich bezeichneten.

Geplante Verbesserungen

Natürlich sahen wir trotzdem einige Möglichkeiten zur Verbesserung. Wir wollten einerseits die Benutzerfreundlichkeit weiter verbessern und andererseits mindestens ein neues, technisch anspruchsvolles Feature implementieren.

Um unsere eigenen Ideen auf ihre Notwendigkeit zu überprüfen sowie uns weiter inspirieren zu lassen, führten wir unter den Studenten der Vorlesung "Betriebssysteme 1" eine Befragung durch, bei der auch Eliane Schmidli und Prof. Richter anwesend waren. Ebenfalls befragt wurde ein Schnupperstudent, der kein Hörer der Vorlesung war und keine formalen Vorkenntnisse in Prozessorarchitektur hatte (dafür aber 5 Jahre Berufserfahrung in der Informatik).

Bei den Befragungen ergaben sich vonseiten der Studenten folgende Vorschläge:

1) Weiterführende Informationen zu den ausgeführten Instruktionen sollten im Simulator verfügbar sein. Das erspart einem das Nachschlagen der x64-Syntax parallel zur Verwendung des Simulators.
2) Eine Möglichkeit, die Animation der einzelnen Schritte zu beschleunigen, fehlt momentan. So könnten Studenten mit fortgeschrittenem Wissen den Simulator verwenden, um ihr Verständnis der Ausführung schnell zu überprüfen, ohne sich durch langwierige Animationen klicken zu müssen.
3) Die Möglichkeit, einen Schritt zurück zu gehen, sollte gegeben sein. So kann man "zurückspulen", wenn einem etwas zu schnell ging oder man etwas nicht ganz verstanden hat.
4) Zwecks Nachvollziehbarkeit wäre es schön, das aktuell ausgeführte Assembly-Programm vor sich zu sehen, während man durch den Simulator klickt. So kann man die aktuelle Instruktion im Kontext des restlichen Programms sehen und besser nachvollziehen, was der Assembly-Code als Ganzes macht. Eventuell könnte die aktuell ausgeführte Animation sogar hervorgehoben werden.
5) Eine Kombination der vorangehenden zwei Punkte: das Klicken auf eine bestimmte Zeile im Assembly-Code bewirkt ein Springen des Programms zu diesem Punkt der Ausführung.

Zu unserer Freude deckten sich diese Vorschläge fast vollständig mit unseren eigenen Ideen. Wir hatten zusätzlich noch folgende Anpassungen am GUI geplant:

6) Prof. Richter hatte uns darauf hingewiesen, dass die Reihenfolge von "Execute Instruction" und "Increment Instruction Pointer" möglicherweise falsch ist. Nach eigener Recherche konnte wir dies bestätigen: moderne Prozessoren "fetchen" zuerst die neue Instruktion, erhöhen dann den Instruction Pointer (IP) um die Länge der soeben erhaltenen Instruktion und führen sie anschliessend erst aus. Daher wollten wir diese Reihenfolge auch im Simulator anpassen.
7) Die Animation der Jump-Befehle wurde von unseren Vorgängern in deren Vorarbeit als unfertig beschrieben, da sie der Standard-Animation (die für alle Befehle verwendet wird) entspricht. Dadurch wird nicht klar, dass ein Jump-Befehl den Instruction Pointer modifiziert. Wir wollten dies durch eine speziell angepasste Animation klar machen.
8) Eine Anpassung des Farbschemas. Das alte Farbschema wirkt etwas knallig und überlastet das Auge mit vielen verschiedenen Farben. Dies wollten wir ändern, ohne dabei die Bedienbarkeit für Leute mit Sehbehinderungen zu vermindern.

Prof. Richter war mit all diesen Änderungen einverstanden und hub insbesondere den "Reverse Debugger" bzw. die Möglichkeit, einen Schritt zurück zu gehen, als primäres technisches Merkmal der Arbeit hervor.

Daher beschlossen wir, alle Ideen zu implementieren. Aufgrund ihrer (vermuteten) technischen Komplexität sowie zeitlichen Restriktionen wurden allerdings Punkt 5 sowie das Hervorheben der aktuellen Code-Zeile als optional eingestuft und niedrig priorisiert.

Benutzeroberfläche

Wie im Projektplan (siehe Anhang «administrativer Bericht») ersichtlich, beschlossen wir, uns anfangs auf das GUI zu konzentrieren und unseren ersten Implementations-Milestone diesem Thema zu widmen.
Dafür gab es mehrere Gründe:

- Es erlaubte uns einen einfacheren Einstieg in den Code. Da die zu implementierenden Features im GUI alle eher klein und oberflächlich waren, war dafür kein tiefes Verständnis des Codes vorausgesetzt.
- Änderungen am GUI sind logischerweise sofort sichtbar und haben oft den grössten Effekt auf die Benutzerfreundlichkeit. Selbst wenn der technisch anspruchsvollere Teil später fehlschlagen sollte, hätten wir durch die fertigen Modifikationen der Benutzeroberfläche somit trotzdem etwas erreicht.
- Die Komplexität der meisten GUI-Änderungen wurde von uns als klein eingeschätzt, was das Risiko reduzieren sollte, unerwartet mehr Zeit zu benötigen. Ausserdem erschienen uns die Aufgaben kleiner, zahlreicher und fast immer voneinander unabhängig - dies würde es einfach machen, bei Zeitmangel bestimmte Features zu streichen, um im Projekt weiter zu kommen.

Animationsgeschwindigkeit

Als (für den Anfang) wichtigstes GUI-Feature erschien uns eine Möglichkeit zum Erhöhen der Animationsgeschwindigkeit. Da das primäre Zielpublikum Studenten sind, deren Wissensstand sich durch den fortlaufenden Konsum der Vorlesung sukzessive erhöht, muss sich auch unser Programm an dieses steigende Wissen anpassen.

Anfänglich ist der Verwendungszweck des Simulators reines Ausprobieren und Nachvollziehen der ablaufenden Animationen. Dabei sind langsam ablaufende Animationen von Vorteil. Später jedoch wird der Simulator zunehmend als Werkzeug benützt, um den Ablauf spezifischer Instruktionen zu untersuchen oder um generell den Effekt eines nicht vollständig verstandenen Assembly-Programms nachzuvollziehen. Dazu ist einerseits das Überspringen von Animationen von Vorteil - dies wurde auch von unseren Vorgängern bereits erkannt, die die Funktion deshalb eingebaut haben. Sie ist hauptsächlich bei den Schritten "Get Instruction" und "Increase Instruction Pointer" nützlich, da diese immer gleich ablaufen. Andererseits ist das Beschleunigen von Animationen nützlich, wenn zum Beispiel verifiziert werden soll, ob die tatsächliche Ausführung der eigenen

Erwartung entspricht. Fehlt diese Funktionalität, wird das Verwenden des Simulators schnell zur Zeitverschwendung, da man hauptsächlich auf das Ablaufen von zu langsamen Animationen wartet.

Hier wäre ein Regler zum Ändern der Geschwindigkeit also sinnvoll. Weiterhin erleichtert er auch uns selbst die Arbeit am Simulator, da wir dann feiner granuliert debuggen können (bspw. den Ablauf von Animationen).

Die Implementierung war denkbar einfach: das bereits verwendete GUI-Framework Qasar bietet ein Slider-Element, das wir in die Oberfläche eingefügt haben. Dadurch lässt sich ein Geschwindigkeits-Wert setzen. Anfänglich hatten wir hier Zahlen unter 0 gewählt (bspw. «0.5», damit die Animationen nur halb so lange dauerten). Auf Rat von Prof. Richter hin haben wir dies jedoch angepasst, sodass man tatsächlich die Abspielgeschwindigkeit der Animation einstellen kann (bspw. 1x, 1.25x oder 2x). Dies wird auch auf Videoplattformen wie z.B. YouTube oder SwitchTube so gehandhabt. Anschliessend werden alle Werte, die die Dauer von Animationen beschreiben, durch diesen Wert dividiert. Da dies das erste Feature war, lag der grösste Zeitaufwand im Verstehen der Architektur und der Frage, wie ein Wert zwischen den verschiedenen Architektur-Ebenen propagiert werden kann.

Erweiterte Tooltips

Das mit Abstand am öftesten geforderte Feature war eine textuelle Beschreibung der aktuell ausgeführten Instruktion. Die Studenten werden in der Vorlesung mit einer großen Anzahl Assembly-Instruktionen konfrontiert und führen anfangs nicht selbst geschriebene Assembly-Programme aus. Dadurch kommt es vor, dass sie auf unbekannte Instruktionen treffen oder die genaue Funktionsweise einer Instruktion nicht kennen.

Ein Tooltip erschien uns als beste Möglichkeit, dieses Problem zu lösen: Tooltips erscheinen nur beim Mouseover, was die visuelle Informationsüberladung für Nutzer, die sie nicht benötigen, deutlich reduziert. Sie sind zudem intuitiv zugänglich - wer mehr über etwas wissen möchte, bewegt oft die Maus dorthin. Weiterhin ist die technische Komplexität der Implementation niedrig.

Die NASM-Assembly-Sprache beinhaltet hunderte verschiedene Instruktionen. Da wir unmöglich jede einzelne beschreiben konnten und es beim Grossteil eher unwahrscheinlich ist, dass die Studenten sie benützen würden, haben wir uns auf die gängigsten und vor allem die in der Vorlesung

verwendeten Instruktionen beschränkt und nur für diese eine Beschreibung verfasst. Für alle anderen Instruktionen bleibt der Tooltip leer.

Tabelle 1 - Implementierte Instruktions-Beschreibungen

Arithmetisch	Jumps	Stack	Logik	Spezial
mov	jmp	push	and	lea
add	je, jne	pop	or	
sub	js, jns	call	xor	
imul	jg, jge	ret	not	
neg	jl, jle			
inc	ja, jae			
cmp	jb, jbe			
sal, sar, shr				

Ausserdem haben wir eine kurze Beschreibung für jedes der im Editor-Bildschirm angezeigten Beispiel-Assembly-Programme verfasst, um deren Funktion klar zu machen.

Code Viewer

Jede normale IDE zeigt während des Debuggings den ausgeführten Code an. Normalerweise wird dabei auch die aktuell ausgeführte Instruktion farblich hervorgehoben.

Unser Ziel war es, ähnliches Verhalten im Simulator zu ermöglichen. Der erste Schritt war hierfür das Anzeigen des Assembly-Codes. Dies war bereits teilweise implementiert: der auszuführende Assembly-Code wird via HTML Link Encoding von der Editor-Seite auf die Simulator-Seite gesendet. Damit wird dann einerseits die Unicorn-Engine und andererseits ein Button zur Anzeige des aktuell ausgeführten Codes beliefert. Letzterer bietet allerdings kein Syntax Highlighting und unterscheidet sich visuell stark vom Editor auf der ersten Seite, sodass für manche Studenten der Zusammenhang möglicherweise nicht klar wäre. Daher haben wir den Button entfernt und

stattdessen seitlich des Memorys dasselbe Editor-Element eingefügt, das auch auf der Startseite zu sehen ist.

Da dies zu Platzproblemen bei der Anzeige führte, haben wir ausserdem die Breite anderer Elemente verkleinert und durch eine JS Media Query sichergestellt, dass der neue Code Viewer nur bei ausreichend breiten Bildschirmen angezeigt wird.

Für den GUI-Milestone war diese Implementation ausreichend. Allerdings besteht hier noch keinerlei logischer Zusammenhang zwischen ausgeführten Instruktionen und angezeigtem Programmcode - letzterer ist einfach nur Text.

Farbschema, Theme Switcher

Die Überarbeitung des Farbschemas war von Anfang an geplant und insbesondere ein Anliegen von Prof. Richter. Nach genaueren Überlegungen kamen wir zu dem Schluss, dass wir nicht nur die Farben verändern wollten, sondern auch deren Einsatzbereich. Das alte Schema beinhaltete eine hohe Anzahl verschiedener Farben, was zu einem uneinheitlichen Erscheinungsbild führte und das Auge dadurch strapazierte. Unser erster Schritt war also eine Vereinheitlichung der Farben von acht auf drei Hauptfarben. Konkret sollten alle Hauptelemente des GUI (CPU, Memory, Code Viewer) dieselbe Hintergrundfarbe besitzen (genannt "Primärfarbe"). Elemente innerhalb dieser Hauptelemente nannten wir Sekundärelemente, sie erhalten die sogenannte "Elementfarbe". Eine weitere Farbe wird für interaktive GUI-Elemente (Buttons, Regler, Toggles) verwendet, sie wird gemäss dem Quasar-Framework "QuasarSecondary" genannt. So ist für den Benutzer weiterhin klar ersichtlich, welche Elemente klickbar sind und zur Bedienung verwendet werden können.

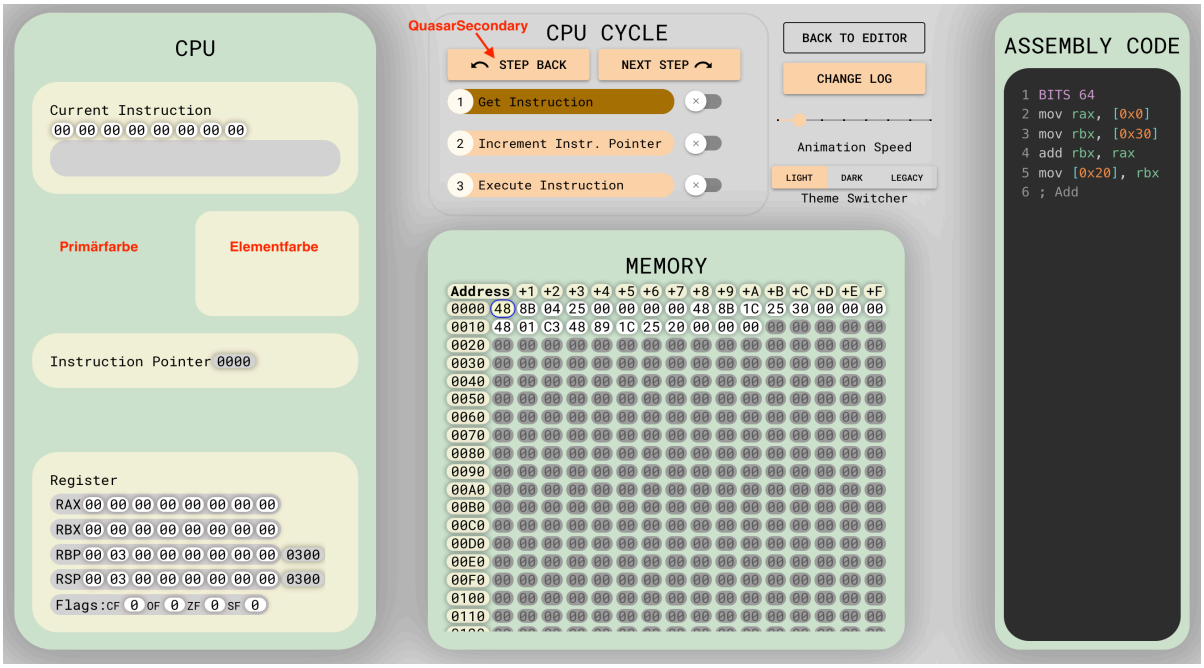

Abbildung 2 - Das "Light Theme" mit Anmerkungen zu den verwendeten Farben

Während des Redesigns kam bei uns eine Idee wieder auf, die wir bereits in der Anfangsphase des Projekts hatten: die Implementierung eines Nachtmodus ("Dark Mode"). Dieser sollte insbesondere für Studenten mit nächtlichem Lernrhythmus äusserst angenehm sein. Da wir sowieso gerade an Farbschemata arbeiteten, beschlossen wir, auch diese Idee umzusetzen.

Ein Farbschema ist letzten Endes natürlich Geschmackssache. Da wir die Arbeit unserer Vorgänger an ihrem Farbschema sehr schätzen, besonders im Bezug auf Bedienbarkeit für Personen mit Sehbehinderung, wollte wir es in unserer Überarbeitung nicht einfach zerstören. Daher beschlossen wir spontan, unsere Änderungen nicht destruktiv durchzuführen und stattdessen einen Theme Switcher einzubauen, mithilfe dessen zwischen einem hellen, einem dunklen und dem bisherigen Farbschema gewechselt werden könnte.

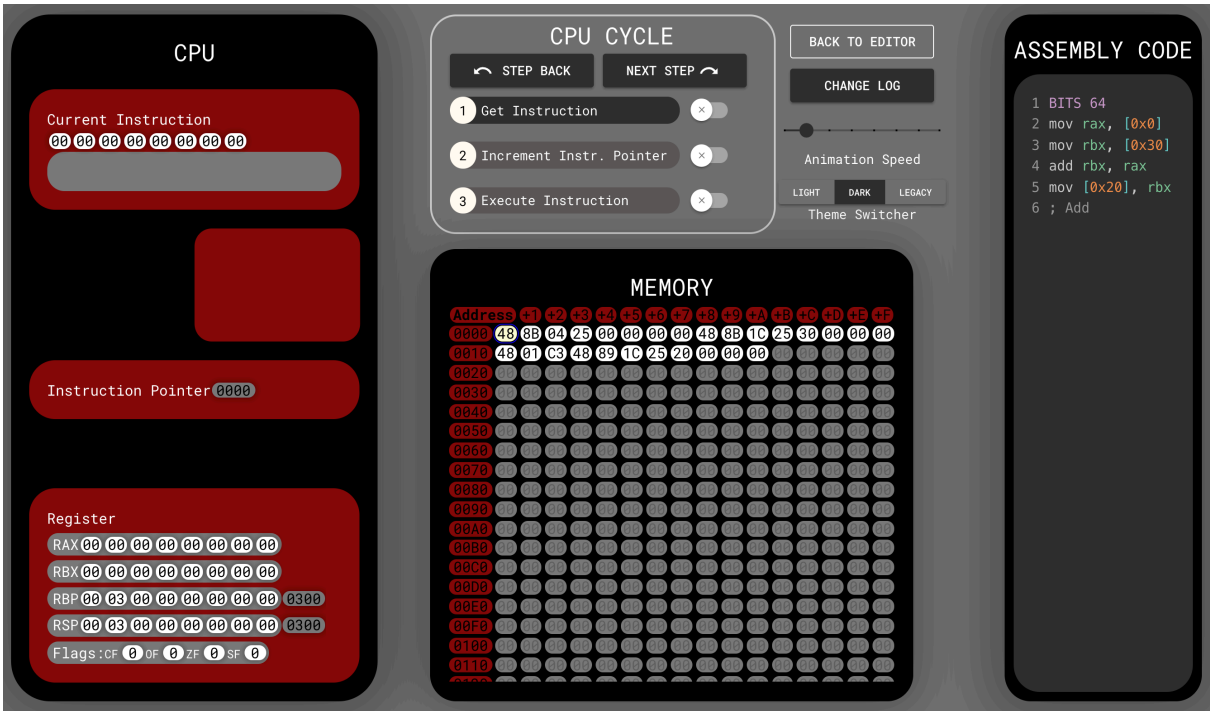

Abbildung 3 - Das "Dark Theme"

Dazu mussten wir die aktuelle Farbstruktur zuerst verstehen und modifizieren. Dabei wollten wir natürlich die Rückwärtskompatibilität zum bisherigen Farbschema aufrecht erhalten. Ausserdem mussten wir zueinander passende Farben für die zwei neuen Farbschemata finden. Zu guter Letzt haben wir einen «Quasar-Toggle» zum Umschalten der Farben eingebaut und unter den anderen Controls platziert.

Da bereits unsere Vorgänger grossen Wert auf gute Lesbarkeit gelegt haben, haben wir unsere beiden neuen Darstellungen mithilfe des Browser-Plugins „Colorblindly“ [1] auf ihre Lesbarkeit für Personen mit Sehschwächen hin überprüft. Ausserdem benutzten wir die Developer Tools von Google Chrome, die einen numerischen Kontrastwert anzeigen, wenn Text gerendert wird. Bei unserem Light Theme liegt dieser Wert nie unter 14,87. Im Dark Theme ist er mit 4,87 deutlich geringer, aber immer noch gut lesbar. Das Legacy Theme hat einen Tiefwert von 7.

Beim Laden der App fragen wir neu ab, ob der Browser gerade ein Dark Theme verwendet oder nicht und passen unser initial gesetztes Farbschema an diese Userpräferenz an. Somit wird verhindert, dass User in der Nacht durch das helle Farbschema geblendet werden, bevor sie eine Möglichkeit bekommen, dieses umzuschalten.

Modifikation der Jump-Animation

Motivation

Die Animation der Jump-Befehle wurde von unseren Vorgängern in deren Vorarbeit als unfertig beschrieben, da sie der Standard-Animation (die für alle Befehle verwendet wird) entspricht. Da bei einem Jump-Befehl aber gegebenenfalls der Instruction Pointer modifiziert wird, stimmt vor allem die anschliessende Animation für den Schritt "Increment Instruction Pointer" nicht. Es entsteht der Eindruck, als würde der Instruction Pointer zuerst von Jump modifiziert, nur um gleich darauf von „Increment“ wieder überschrieben zu werden. Da wir die Reihenfolge des "Increment IP" sowieso anpassen wollten, löst dies einen Teil des Problems. Allerdings ist immer noch nicht klar, dass der IP von Jump überhaupt modifiziert wird, da keine Bytes in den IP-Register fliegen. Daher wollten wir eine andere Animation für Jump-Befehle erstellen, die genau dies zeigt.

Vorgehen

In einem ersten Schritt wollten wir uns auf eine gemeinsame Animation einigen, die das Anpassen des Instruction Pointers bei beispielsweise Jump- und Call Instruktionen am besten abbildet. Da keine der bereits existierenden Animationen ausreichend war, beschlossen wir, eine eigene Lösung zu entwerfen.

Unsere Umsetzung setzte sich zum Ziel, möglichst verständlich zu sein, ohne zusätzliche Komplexität einzuführen. Nach einigen Besprechungen und Prototypisierungen haben wir uns für die folgende Variante entschieden:

Die Komponente *ExecutionBox.vue* soll um ein zusätzliches Feld «Destination» erweitert werden. Dieses Feld ist auf der rechten Seite innerhalb der Box ersichtlich, sobald eine Jump- oder Call Instruktion ausgeführt wird, die auch den Instruction Pointer modifiziert. Wenn es sich um eine solche Instruktion handelt, der Instruktion Pointer aber nicht verändert wird, weil z.B. die Bedingung für den Sprung nicht erfüllt wird (z.B. bei ‘jne’ oder ‘je’ möglich), wird «Destination» nicht eingeblendet.

Die «Destination» entspricht im Fall eines Sprungs der Zieladresse im Memory. Es folgt eine visuelle Darstellung der eben beschriebenen Änderungen.

Visualisierung

Die nachfolgenden Grafiken visualisieren den Schritt «Execute Instruction» der Instruktion «jne» (= **j**ump (if) **n**ot **e**qual). Es handelt sich hier um einen konditionalen Sprung (Jump), der effektiv ausgeführt wird. Diese Voraussetzungen führen zum Abspielen unserer neuen Animation:

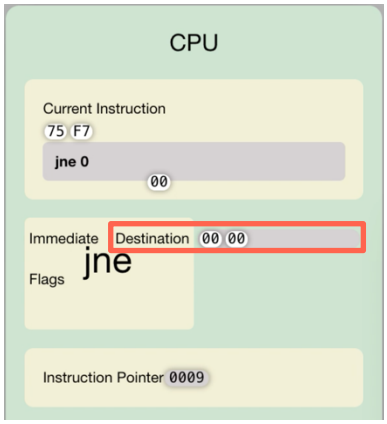

Abbildung 4

0. «Destination» wird neu eingeblendet, da gesprungen wird (ZF=0). Die Bytes der Zieladresse sind rechts dargestellt und definieren den neuen Instruction Pointer.

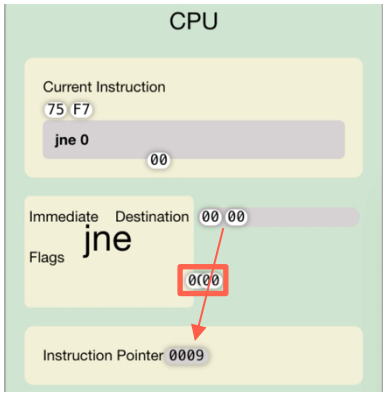

Abbildung 5

1. Die Bytes «fliegen» in Richtung des Instruction Pointers, weil der Wert in «Destination» (00 00) in dieses Register geladen wird.

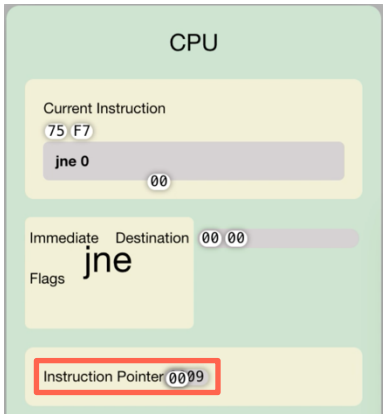

Abbildung 6

2. Die Bytes erreichen den Instruction Pointer und überschreiben diesen mit der neuen Adresse.

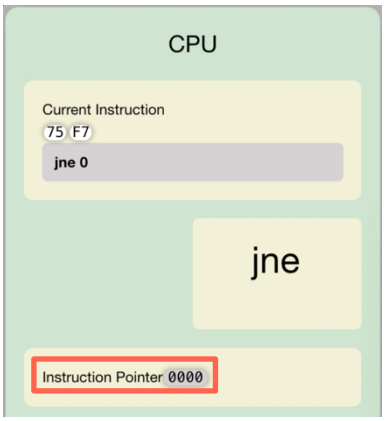

Abbildung 7

3. Der Pointer ist nun auf dem aktuellen Stand und zeigt auf die korrekte Adresse, die Animation ist abgeschlossen.

Implementierung

Die Implementierung dieses neuen Features nahm verhältnismässig viel Zeit in Anspruch. Es musste ein Querschnitt durch die ganze Applikation vorgenommen werden - von der eingebundenen Unicorn-Instanz bis zum GUI, dass sich anpassen sollte.

Der spannendste Aspekt an dieser Lösung ist die Logik, um festzustellen, ob der Instruction Pointer durch eine Instruktion erhöht wird. Zunächst hatten wir vor, die Instruktionen manuell darauf zu überprüfen, ob sie den IP manipulieren. Wir merkten allerdings schnell, dass wir diese Variante auf jeden Fall vermeiden wollten. Der Code wäre aufgrund der vielen «if/else» oder «Switch»-Fälle nicht nur unschön anzusehen, sondern hätte sicherlich auch zu enormen Performance-Einbussen geführt. Ausserdem merkten wir zum Schluss, dass dadurch die eigentliche Frage (ob denn jetzt gesprungen wird oder nicht) gar nicht beantwortbar wäre. Daher entschieden wir uns, diese Idee zu verwerfen und machten uns auf die Suche nach einer anderen Lösung.

Der gesamte Prozessorsimulator basiert im Kern auf der Unicorn-Engine, die unsere Instruktionen verarbeitet und uns die Resultate nach jeder Operation zurückgibt. Dieses Emulations-Framework ermöglichte uns deshalb auch herauszufinden, ob der Instruktions-Pointer verändert wurde. Handelt es sich um eine Veränderung, die nicht mit einer regulären Erhöhung des IP zu erklären ist, handelt es sich offensichtlich um eine solche Instruktion. Dieses Verhalten nahmen wir uns zu Hilfe, um die Logik zu entwerfen, die für die Grundlage der Animation zuständig ist. Im nächsten Kaptiel «Neuer Programmablauf» wird auf die Berechnung und Unterscheidung der Fälle eingegangen.

Neuer Programmablauf

Motivation

Während einer Besprechung mit Prof. Richter kam das Thema des Programmflusses auf. Wie bereits erwähnt funktionieren aktuelle Prozessorarchitekturen nach dem folgenden Schema:

- Get Instruction -> Increment Instruction Pointer -> Execute Instruction

Die Reihenfolge im Simulator war zum Zeitpunkt unserer Besprechung aber eine andere, nämlich:

- Get Instruction -> Execute Instruction -> Increment Instruction Pointer

Wir wollten das Verhalten richtigstellen und entschieden uns dafür, die Reihenfolge der Schritte anzupassen.

Vorgehen

Die initialen Aufgaben fielen uns leicht. An vielen Stellen im Code musste lediglich die Reihenfolge der Schritte innerhalb von Arrays angepasst werden:

```
private steps: Array<CpuCycleStep> = [{
  name: 'Get Instruction',
  numberInCycleSequence: Step.GET_INSTRUCTION,
  animate: false,
}, {
  name: 'Increment Instr. Pointer',
  numberInCycleSequence: Step.INCREASE_IP,
  animate: false,
}, {
  name: 'Execute Instruction',
  numberInCycleSequence: Step.EXECUTE_INSTRUCTION,
  animate: false,
}];
```

Abbildung 8 - Step-Arrays

Das User-Interface wird auf Basis dieses Arrays aufgebaut und benötigte deshalb keine zusätzlichen Änderungen.

Uns stellte sich jedoch beim weiteren Vorgehen ein Problem. Die Erhöhung des Instruction Pointers (IP) wird dynamisch aus *Unicorn*, der Emulations-Engine, geladen. Diese liefert uns lediglich den endgültigen Wert des IP, also den Wert nach Ausführen *aller* Schritte der Instruktion.

Generell kann jedoch nicht davon ausgegangen werden, dass dieser finale Wert, der aus Unicorn geladen wird, der Erhöhung im Schritt „Increment

Instruction Pointer“ entspricht. Die im vorherigen Kapitel besprochenen Jump/Call-Instruktionen modifizieren dieses Register nämlich ebenfalls. Die von unseren Vorgängern implementierte Variante setzt beim letzten Schritt «Increment Instruction Pointer» den Wert einfach direkt auf den neuen Zielwert, egal ob dieser durch eine Instruktion verändert wurde oder einer «natürlichen» Erhöhung entsprach.

Um die Prozessabfolge in die von uns gewünschte Reihenfolge zu bringen, mussten wir den neuen Wert des IP von Hand berechnen, wann immer dieser durch «Increment» erhöht wurde. Nur dadurch würden wir im Zeitraum zwischen «Increment» und „Execute“ den korrekten Wert anzeigen können. Diese Funktionalität existierte bis anhin noch nicht.

Die durch «Increment» bedingte Erhöhung des Registers RIP entspricht der Länge der Instruktion in Bytes. Via dem bereits eingebundenen Disassembler «Capstone» erfahren wir die Länge der aktuellen Instruktion. Wir erhöhen den Wert also manuell um diese Länge und erreichen somit unser Ziel.

Die Applikation funktionierte nach diesen Anpassungen wie gewünscht. Allerdings mussten noch fast alle Test-Cases umgeschrieben werden, da wir die erwarteten Werte im Feld des Instruction Pointers an die neuen, tatsächlichen Werte anpassen mussten.

	Schritt 0	**Schritt 1**	**Schritt 2**
Vorher	**Get Instruction:** Liest die aktuelle Instruktion aus dem Memory.	**Execute Instruction:** Animiert die Evaluation des jumps, modifiziert den IP jedoch nicht (auch wenn gesprungen wird)	**Increment Instruction Pointer:** Erhöht den IP (um die Länge der Instruktion oder auf neue Zieladresse)
Jetzt	**Get Instruction:** (Analog)	**Increment Instruction Pointer:** Erhöht den IP in jedem Fall um die Länge der Instruktion.	**Execute Instruktion:** Verarbeitet die Instruktion und modifiziert möglicherweise den Instruktion Pointer erneut. (JMP/CALL/RET, etc.)

Reverse Debugging (Time-Travel debugging)

Literaturrecherche

Vor dem Beginn unserer Arbeit am Reverse Debugger haben wir uns bezüglich des aktuellen Stands der Forschung zu diesem Thema informiert. Im Paper „A Review of Reverse Debugging“ [2] gibt Jakob Engblom einen Überblick über die Begrifflichkeiten und fasst die verbreitetsten Implementationsmethoden zusammen.

Weiterhin konnten wir in "Implementation of Live Reverse Debugging in LLDB" [3] von Savidis & Tsiatsianas eine genaue technische Beschreibung zur Implementation eines Reverse Debuggers für LLDB finden.

Die meisten anderen Paper oder Webseiten zu dem Thema befassen sich mit den Vorteilen von Reverse Debugging bezüglich Multithreading und der Detektion sogenannter *Heisenbugs* (Bugs, deren Auftreten durch das Starten einer Debugging-Session beeinflusst wird) und waren daher für unsere Arbeit weniger bedeutsam. [4-6]

Das grundlegende Problem beim Implementieren eines rückwärts laufenden Debuggers ist, dass der vergangene Zustand eines Computersystems sich nicht immer vollständig aus dem aktuellen Zustand ableiten lässt. Dies gilt auch für unseren Simulator. Es existieren eine Vielzahl von Operationen, die Informationen unwiderruflich zerstören (ein Beispiel: `xor rax rax`). Jeder Reverse Debugger muss daher extern Informationen zwischenspeichern. Hierfür gibt es zwei grundlegende Möglichkeiten:

- **Record-Replay**: eine Ausführung des Programms wird aufgezeichnet, inklusive aller nicht-deterministischen Events (bspw. User-Input, Netzwerkaktivität, etc.). Bei Bedarf kann also diese bestimmte Ausführung des Programms von Anfang an wiederholt und somit erneut debuggt werden. Es ist allerdings nicht möglich, einzelne Schritte zurück zu gehen. Die rückwärtige Ausführung zu Reverse Breakpoints ist ebenfalls nicht möglich.
  - **Trace-based debugging**: eine Spezialform des Record-Replay-Debuggings, bei dem der gesamte Systemzustand aufgezeichnet wird. Meist nur durch Hardware-Support möglich.
- **Reconstructive**: es werden zu bestimmten Zeitpunkten komplette Abbildungen des Systems erstellt. Wünscht man nun in der Exekution nach hinten zu laufen, so setzt sich das System auf einen dieser vergangenen Zustände zurück und läuft von dort aus vorwärts bis zum

gewünschten Zeitpunkt. Diese Variante ist relativ flexibel: beispielsweise kann durch die Häufigkeit der "Snapshots" die Granularität und der resultierende Aufwand beim Zurückspringen gesteuert werden. Je mehr Snapshots erstellt werden, desto mehr Speicher wird beansprucht, dafür ist die Rechenlast beim Zurückspringen geringer. So kann Speicher gegen CPU-Leistung abgewogen werden. Ausserdem könnten die Snapshots beispielsweise auch inkrementell erstellt werden, sodass immer nur die Daten abgespeichert werden, die sich verändern und nicht jedes Mal das gesamte System. Wichtig ist, dass auch hier nicht-deterministische Inputs aufgezeichnet werden müssen.

Ansatz 1: komplette Aufzeichnung

Motivation

Inspiriert von der «Reconstructive» - Lösung in der Literatur versuchten wir als ersten Ansatz, den kompletten Zustand unseres Systems aufzuzeichnen und temporär zu speichern. Die Idee dahinter war, ein Gefühl für die anfallende Datenmenge und somit den Speicher-Aufwand zu bekommen. Ausserdem wollten wir ausloten, ob es prinzipiell möglich wäre, den Systemzustand abzuspeichern oder ob uns irgendetwas daran hindern würde.

Unser erster Prototyp sollte also in der Lage sein, Schritte rückwärts zu gehen, indem ein gesamter vergangener Systemzustand aus dem Speicher geladen wird und von dort an die Ausführung fortsetzt.

Implementation

Als erstes wollten wir in Erfahrung bringen, welche Objekte für den «State» der Applikation relevant sind. Wir stiessen auf drei Kandidaten:

Objekt	**currentStep**	**Program**	**State**
Beschreibung	Beinhaltet Informationen zum momentanen Schritt innerhalb der Simulation (siehe «neuer Programmablauf»).	Beinhaltet Objekte, die für Interaktionen mit *Unicorn* benötigt werden.	Beinhaltet Objekte, die relevant für das GUI sind.

Mit Hilfe des Debuggers von Webstorm und der Analyse des Codes innerhalb des Controllers stellte sich heraus, dass diese drei Objekte für eine vollständige Wiederherstellung des Zustands genügen würden. Würden wir sie also vor oder nach jedem Schritt des Simulators zwischenspeichern, sollte ein Zurückschreiten im Programmfluss möglich sein.

Um die sich verändernden Daten in jedem Schritt zu speichern, erstellten wir ein neues *StateStore*-Objekt (siehe Abbildung 9). Dieses beinhaltet ein Array von *StateContainer*-Objekten, wobei für jede Instruktion ein *StateContainer* enthalten ist.

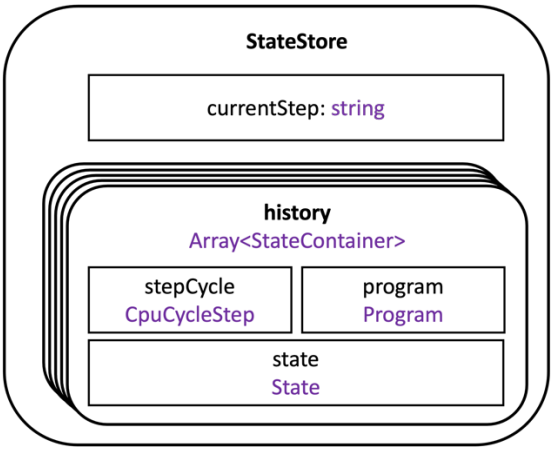

Abbildung 9 - Der StateStore in seiner ursprünglichen Form

In jedem *StateContainer* sind die drei oben genannten Objekte abgelegt. Jeder Eintrag im Array *history* entspricht somit einem Snapshot unseres Programmes.

Der *StateStore* wird mit der Initialisierung des Simulators mit dem Laden des *Controllers* erstellt und bereits mit einem ersten *StateContainer* befüllt. In diesem sind alle Startwerte des Programms enthalten, was dem Zustand der Applikation zum Zeitpunkt null entspricht.

Wird im Simulator ein Schritt nach vorne navigiert, erzeugen wir einen neuen Snapshot zu diesem Zeitpunkt.

Bei einem Schritt rückwärts laden wir den aktuellsten Eintrag in unserem *StateStore* via *LoadStateStore* in den Simulator und löschen den Eintrag anschliessend aus dem *history*-Array - analog zu einer „Pop-Operation“ in einem Stack.

Resultat

Es gelang uns ein erster funktionaler Prototyp. Das Zurückspringen funktionierte, hatte aber einige entscheidende Nachteile. Die Performance war inakzeptabel: der Simulator fror für zwei bis drei Sekunden ein, nachdem man den «Step back» - Knopf betätigt hatte.

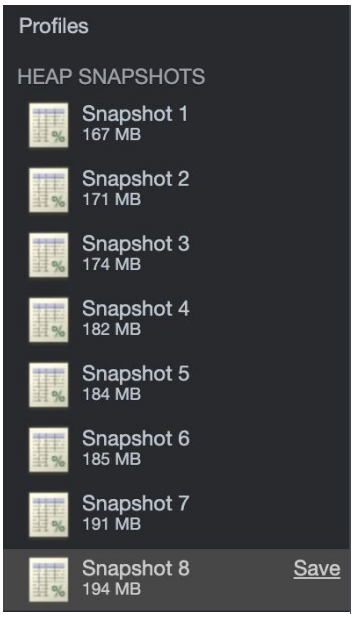

Abbildung 10 - Der Anstieg des benötigten Speichers ist klar ersichtlich

Ausserdem zeigte eine Memory-Analyse, dass der verwendete Speicher linear stark anstieg (2-3 MB pro Schritt), auch wenn man den Simulator ausschliesslich vorwärts laufen liess.

Dies entsprach zwar unseren Erwartungen, war aber durch das mögliche Abspielen von längeren Assembly-Programmen keine dauerhafte Lösung. Als temporärere Umgehung dieses Problems bauten wir deshalb einen «Toggle» im Editor-Bildschirm ein, mit dem sich der Reverse Debugger (und seine speicherintensiven Aufzeichnungen) deaktivieren liess.

Ansatz 2: Transaktions-Analyse

Idee & Entscheidungsfindung

Der erste Ansatz hatte also bewiesen, dass unser Vorhaben in der Theorie durchführbar war. Ausserdem hatten wir wertvolle Erkenntnisse darüber gesammelt, welche Daten genau gespeichert werden mussten, um das System in einen früheren Zustand zurückzusetzen. Unser nächstes Ziel war daher, die Performance zu verbessern sowie den Speicheraufwand zu reduzieren.

Wir teilten uns daher auf: während einer von uns versuchte herauszufinden, weshalb das «Reversieren» so starke Performance-Einbussen mit sich brachte, versuchte der andere einen neuen Ansatz: eine **Transaktionsanalyse**. Die Idee war simpel: wir würden lediglich alle *Änderungen*, die in einem Schritt am *State* vorgenommen werden, aufzeichnen. Im Gegensatz zu Ansatz 1 würde dies deutlich weniger Speicher verwenden, da Felder nicht redundant aufgezeichnet werden würden, sondern nur, wenn sie sich tatsächlich verändern. Trotzdem wären wir jederzeit in der Lage, einen vergangenen Zustand wiederherstellen zu können, indem wir die veränderten Felder neu beschreiben.

Wir hatten schon anfangs mit dieser Idee gespielt und wollten sie auf ihre Umsetzbarkeit überprüfen. Allerdings schreckten uns zwei Umstände ab: erstens, dass wir nicht wussten, mit wie vielen Daten (und in welchen Strukturen) der Simulator arbeitete. Dies hatte sich aber durch die Implementierung von Ansatz 1 geändert. Zweitens kommt in der von uns studierten Literatur eine solche veränderungsbasierte Lösung nicht vor. Wir vermuten, dass ein vollwertiges, allgemeines Computerprogramm zu grosse Mengen an Daten verändert und zu viele nicht einfach reversierbare Nebeneffekte auslöst, als dass sich einzelne Instruktionen einfach «zurückrollen» liessen. Vor allem nicht deterministische Inputs wie User-Interaktionen, File-System-Reads, Netzwerkaktivität oder parallel interagierende Threads sind vermutlich äusserst problematisch.
Unser Simulator ist allerdings deutlich einfacher gestrickt. Im Prinzip müssen wir lediglich den Inhalt der Register, die Flags sowie das Memory auf Änderungen überwachen. Nicht-deterministische Inputs gibt es keine. Der Ablauf der Simulation ist mit abgeschlossener Eingabe des Assembly-Programms vorhersehbar (natürlich nur innerhalb gewisser Grenzen [7]).

Inzwischen hatten wir auch herausgefunden, was beim kompletten Wiederherstellen des Programmzustands zu den starken Performance-Einbussen führte. Da wir den ganzen Speicherbereich wiederherstellen mussten, führte dies aufseiten des Browsers zu einem erneuten Rendern aller Memory-Bytes. Da es 4.096 davon gibt und jede davon ihre eigene Vue-Komponente mit eigener Logik ist, dauerte dieser Prozess relativ lange (eben die oben genannten zwei bis drei Sekunden). Während dieser Zeit war das GUI nicht verwendbar, die Applikation reagierte nicht.

Wir fanden nach einiger Überlegung keine wirkliche Möglichkeit, wie wir das Ersetzen des gesamten Speichers auf effiziente Art vermeiden konnten. Wir dachten zunächst an einen Vergleich («Diff») des Speicherinhalts mit dem vergangenen Zustand und ein darauffolgendes selektives Bearbeiten einzelner Bytes. Allerdings erschien uns dies als eine sehr aufwändige Operation – vor allem in Hinblick darauf, dass sie bei jedem einzelnen Schritt (zurück) durchgeführt werden müsste. Die einzige Möglichkeit, den Vergleich zu vermeiden, wäre die oben überlegte Transaktionsanalyse. Dies bestärkte uns zusätzlich in unserem Vorhaben, diese zu implementieren. Erneut beschlossen wir, uns aufzuteilen, um getrennt nach der besten Lösung zu suchen. Bevor wir die beiden Ansätze jedoch erklären, ist ein kurzer Blick auf den internen Ablauf des Simulators notwendig.

Ablauf der Simulation

Der Simulator arbeitet mit einem Zustandsobjekt (genannt *State*), das sowohl die Inhalte des Speichers als auch die der Register abbildet. Aufgrund einer unsauberen Trennung zwischen Modell und View wird dieser *State* im Zuge der GUI-Animationen zusammen mit dem GUI aktualisiert. Die neuen Daten für diese Aktualisierung kommen dabei aus dem **Unicorn-Emulator**, ein externes Tool, das eine Assembly-Instruktion als Input nimmt und den Zustand von Registern und Speicher nach der Ausführung dieser Instruktion zurückgibt. Der Emulator hat dabei seinen eigenen, internen Zustand, der für den Simulator nicht sichtbar ist und ausschliesslich vom Emulator selbst modifiziert wird.

Der Ablauf der Simulation ist unterteilt die Ausführung verschiedener Assembly-**Instruktionen**, die der Benutzer im Editor eingibt. Wird im Editor der Button «Start Program» gedrückt, wird der Assembly Code an den Simulator übergeben. Dieser lädt eine Instanz des Unicorn-Emulators mit leeren Registern und dem auszuführenden Programm im Speicher. Anschliessend führt der Simulator durch GUI-Interaktionen drei **Schritte** pro Instruktion aus. Es folgt eine Beschreibung dieser Schritte, wobei der interne Zustand des Simulators mit [Simulator] und der des Emulators mit [Unicorn] gekennzeichnet ist.

1) GET INSTRUCTION
   [Simulator]: der Simulator aktualisiert die *current Instruction* in seinem *State*. Im GUI wird eine diese in Textform angezeigt und das Laden aus dem Speicher animiert. Der soeben geladene Speicherbereich wird visuell hervorgehoben.
   [Unicorn]: auf technischer Seite findet in diesem Schritt die Ausführung der aktuellen Instruktion im Unicorn-Emulator statt. Wichtig ist hierbei anzumerken, dass Unicorn alle drei Schritte (GET, INCREMENT EXECUTE) auf einmal ausführt und alle sich verändernden Felder in seinem internen State bereits hier neu setzt. Ein Objekt «accessedElements», das diese veränderten Felder beinhaltet, wird an den Simulator zurückgegeben (dazu später mehr). Das GUI zeigt die durchgeführte Instruktion jedoch an dieser Stelle noch nicht – dies geschieht erst in den nächsten Schritten.

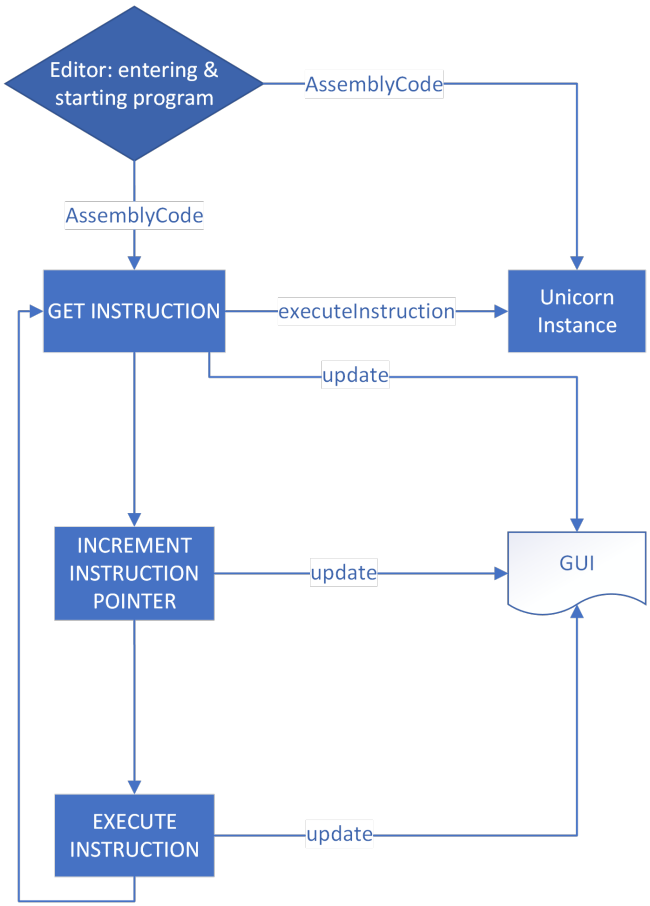

Abbildung 11 - Der Ablauf der Simulation

2) INCREMENT INSTRUCTION POINTER
   [Simulator] Der Instruction Pointer (IP) wird um die Länge der gerade geladenen Instruktion erhöht. Dies verändert nur den Instruction Pointer im *State* des Simulators und ist ansonsten eine rein visuelle Animation. In der Unicorn-Instanz ändert sich nichts.
3) EXECUTE INSTRUCTION
   [Simulator] Die Instruktion wird ausgeführt und Daten werden zwischen Speicher und Registern bewegt. Wie bereits INCREMENT ist auch dies lediglich eine Zustandsveränderung im Simulator und ändert nichts an der Unicorn-Instanz.

Anschliessend wird die nächste Instruktion ausgeführt - so lange, bis keine Instruktionen mehr vorhanden sind.

Implementierung: Ansatz A - Datenstruktur

Dies war der Versuch, alle veränderten Daten direkt in eine Datenstruktur abzuspeichern. Dabei wurde die bereits im Simulator vorhandene Struktur *accessedElements* verwendet (siehe Abbildung 12). Diese wird im Schritt «Get Instruction» von Unicorn zurückgegeben und enthält alle Register und Speicherbereiche, aus denen gelesen oder in die geschrieben wurde. Ausserdem sind die gesetzten Flags enthalten. Zu Zwecken des Reverse Debuggings sind für uns lediglich Schreiboperationen sowie gesetzte Flags relevant, da diese den *State* verändern.

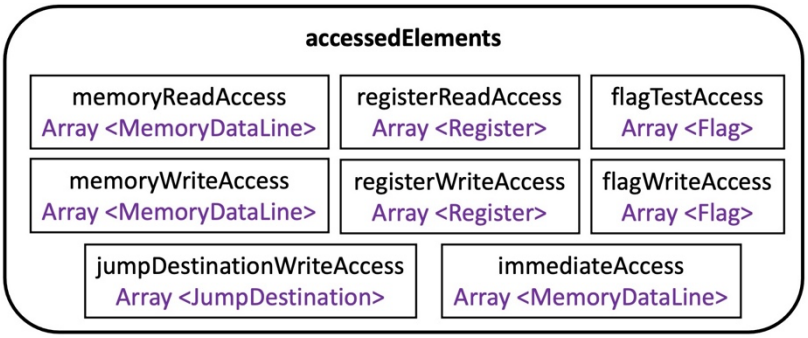

Abbildung 12 - Die Datenstruktur "accessedElements"

Dazu passten wir den StateContainer aus der Reconstruction-Lösung wie in Abbildung 13 ersichtlich an. Wir fügten also das *accessedElements*-Objekt sowie den Instruction Pointer vor und nach seiner Erhöhung durch den Increment-Schritt hinzu. Den IP nach seiner Erhöhung zu speichern ist notwendig, falls ein darauffolgender Jump ihn modifiziert.

Diese Lösung funktionierte, solange man innerhalb einer Instruktion Schritte rückwärts machen wollte. Allerdings bemerkten wir, dass bei einem Reversieren über die aktuelle Instruktion hinaus die Werte nicht korrekt aktualisiert wurden. Uns wurde schnell bewusst, dass dies an Java's «copy by reference» lag, bei dem nur Referenzen auf Objekte kopiert werden anstatt die Objekte selbst. Offenbar hatten wir daher immer dasselbe *accessedElements*-Objekt in der *history* gespeichert.

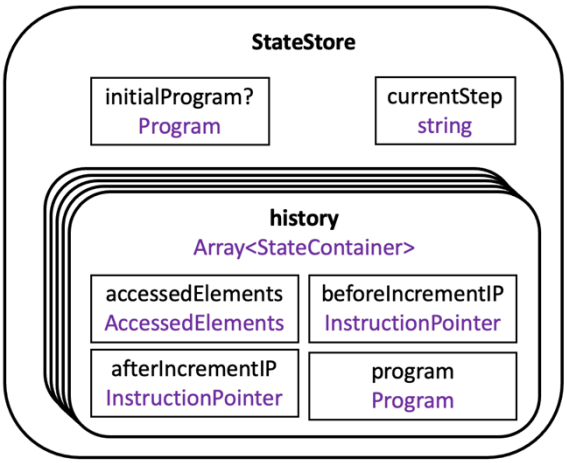

Abbildung 13 - Das StateStore-Objekt

Wir brauchten also einen Weg, die Objekte «by value» zu kopieren. Es stellte sich heraus, dass dies in Javascript nicht so einfach ist, wie gedacht. Natives Javascript bietet keine Möglichkeit dazu. Daher waren wir gezwungen, uns an eine externe Library zu wenden und wurden dabei bei «Really Fast Deep Clone» (rfdc, [8]) fündig. Nun funktionierte zwar das Kopieren der *accessedElements*-Objekte, die Felder wurden jedoch immer noch nicht richtig befüllt. Der Grund dafür war,

dass die rfdc-Library offenbar nicht in der Lage war, die Unicorn-Instanz korrekt zu klonen. Da wir während Implementierungsansatz B auf dasselbe Problem stiessen und es dort aber lösen konnten, führten wir Ansatz A nicht weiter fort.

Implementierung: Ansatz B – Proxies

Wir suchten nach einer Möglichkeit, Änderungen an Objekten festzustellen. Optimal wäre eine Javascript-native Variante. Wir stiessen initial auf *Object.observe*(). Es handelt sich dabei um einen «ECMAScript» Standard, der in der Lage ist, Änderungen an Objekten festzustellen.

Leider ist diese Methode mittlerweile deprecated. Ersetzt wurde Object.observe() durch die sogenannten Proxies. [9]

Proxies sind eine Art von Wrapper, die das darunterliegende Objekt ummanteln und Aktionen auf diesem aufzeichnen, modifizieren oder komplett verwerfen können. Ein Proxy wird via dem *target (*dem Zielobjekt) und einer *handler* Funktion aufgebaut.

```
const target = {
  message: "hello",
}

const handler = {
  get: function(target, property, receiver) {
    return target[property] + " world";
  }
};

const proxy = new Proxy(target, handler);


console.log(proxy.message); // "hello world"
```

Abbildung 14 - Ein Beispiel für einen Proxy

Als Rückgabewert erhalten wir einen Proxy. Dieser Proxy weist gegen aussen die gleichen Eigenschaften, Felder und Methoden aus, wie das ursprüngliche Objekt.

Die dabei interessante Eigenschaft dieser Proxies wird mittels der *handler* Funktion definiert. In dieser können wir mittels Gettern und Settern zusätzliches Verhalten bestimmen. Ein Beispiel dazu ist in Abbildung 14 ersichtlich: immer dann, wenn auf das Feld *message* innerhalb des *proxy*-Objektes zugegriffen wird, erhalten wir die Zeichenkette „hello world“ zurück.

Analog zu *get* sind weitere sogenannte *traps* möglich. *Set, get, has* und *defineProperty* sind nur eine kleine Auswahl davon.

Nach einem kleinen Prototypen innerhalb unserer Applikation waren wir nun sicher, dass wir Änderungen speichern konnten. Die Grundlage für den transaktionsbasierten Ansatz war gelegt.

Einbindung Proxies

Um die Proxies sinnvoll einzubinden erstellten wir zwei Methoden: *attachState(target: State)* und *attachProgram(target: Program)* innerhalb unserer Klasse *Reverse Debugger*. Zur Verdeutlichung demonstrieren wir hier *attachState* (Abbildung 15).

```
private attachState(target: State) {
  const setTransactionStore = (prop: string, value: ValueOfState) => {
    this.setTransactionStore(prop, value);
  };

  const handler = {
    set(obj: State, prop: string, value: ValueOfState) {
      setTransactionStore(prop, value);
      return Reflect.set(obj, prop, value);
    },
  };

  return new Proxy(target, handler);
}
```

Abbildung 15 - attachState

Sobald ein Feld auf State neu gesetzt wird, führt der Handler die Funktion *setTransactionStore* aus. Wir übergeben den Namen des Feldes wie auch den neuen Wert. Der Handler setzt danach via *Reflect.set* das Ursprungsobjekt auf den neuen Wert. [10]

Via *setTransactionStore* wird der neue Wert in die Datenstruktur eingebunden.

Wahl der Datenstruktur

Die im Ansatz 1 gewählte Datenstruktur war nicht mehr ausreichend. Unser Ziel war es, Änderungen pro Feld innerhalb eines Objektes zu speichern und nicht mehr den gesamten Zustand des Objektes.

Das neue Modell trägt die in Abbildung 16 ersichtliche Struktur.

Für jedes Feld innerhalb von *State* legen wir ein Feld in *StateTransactions* an und initialisieren es mit einem leeren Array.

```
interface StateTransactions {
  memoryData: [];
  registers: [];
  currentInstruction: [];
  instructionPointer: [];
  flags: [];
  currentAccessedElements: [];
  byteInformation: [];
  changeHistory: [];
}
```

Abbildung 16 - Das StateTransaction-Interface

Sobald eine Änderung am State vorgenommen wurde, wird der neue Wert im dazugehörenden Feld-Array in *StateTransactions* angehängt.

Versionierung

Die Änderungen konnten also nun dank der Proxies zwischengespeichert werden. Wir haben Werte, die in das Objekt *StateTransactions* geschrieben werden, aber noch keine Möglichkeit diese Werte einem Zeitpunkt im Programmfluss zuzuweisen. Die Lösung für dieses Problem war eine Versionierung.

Um den genauen Zeitpunkt festzuhalten sind zwei Informationen entscheidend: bei welcher Instruktion und in welchem Schritt innerhalb dieser Instruktion wir uns befinden.

```
interface Version {
  step: number;
  nrOfInstructions: number;
}
```

Abbildung 17 - Das Interface 'Version'

Die Information, in welchem Schritt wir uns befinden, erhalten wir aus der Variable *currentStep*, die Teil des aktuellen *State*-Objekts ist. Hier können wir beim Schritt zurück also einfach eins abziehen – somit wird aus „Execute Instruction“ zum Beispiel „Increment Instruction Pointer“. Die *nrOfInstructions* wird mittels der Länge des *currentInstruction*-Arrays, innerhalb von *StateTransactions,* berechnet. Jede Änderung an *currentInstruction* ist einer neuen Instruktion gleichgesetzt.
Der neue Typ *Version* bildet diese beiden Werte ab (siehe Abbildung 17).

Erweiterung von StateTransactions

Wir wollten die Versionierung mit den transaktionellen Änderungen kombinieren. Zu diesem Zweck erweiterten wir unsere Datenstruktur wie in Abbildung 18 ersichtlich.

Ausbau der Abdeckung

StateTransactions speichert alle Änderungen am *State* und versieht jeden Eintrag mit einer eindeutigen *Version*.

```
interface StateTransactions {
  memoryData: [{version: Version; value: MemoryData}];
  registers: [{version: Version; value: Array<Register>}];
  currentInstruction: [{version: Version; value: Instruction}];
  instructionPointer: [{version: Version; value: InstructionPointer}];
  flags: [{version: Version; value: Array<Flag>}];
  currentAccessedElements: [{version: Version; value: AccessedElements}];
  byteInformation: [{version: Version; value: ByteInformation}];
  changeHistory: [{version: Version; value: Array<ChangeHistory>}];
}
```

Abbildung 18 - Das Interface StateTransactions inkl. Versionierung

Wir haben an diesem Punkt zwei der drei notwendigen Komponenten in eine transaktionelle

Datenstruktur eingebunden. Der letzte Bestandteil, der noch fehlt, ist die *Program*-Komponente. In ihr werden die *Unicorn*-Instanz sowie alle dafür relevanten Zusatzfelder abgespeichert.

Leider war es uns nicht möglich, Änderungen an der *Unicorn*-Instanz aufzuzeichnen. Die *Traps*, die die *Proxy* Schnittstelle anbietet, sind nicht ausreichend, um die Veränderungen aufzuzeichnen. Wir vermuten, dass das daran liegt, dass der *Unicorn*-Emulator aus einer externen Library eingebunden wird.

Wir entschlossen uns deshalb für die in Ansatz 1 besprochene Methodik (kompletter Snapshot), um *Unicorn* und die restlichen Daten aus *Program* zwischen zu speichern.

Mit *Unicorn* wird immer dann interagiert, wenn der „Get“ Schritt durchlaufen wird. Dehalb findet unser Snapshot nach „Get“ statt.

Ursprünglich wollten wir einfach die komplette *Program*-Instanz zwischenspeichern. Wie bereits erwähnt trafen wir aber auch hier auf das Problem, dass sich die *Unicorn*-Instanz nicht vernünftig klonen liess. Wir waren aber darauf angewiesen, den alten Zustand von *Unicorn* auslesen zu können, um eine normale, vorwärts laufende Exekution nach einem Rückwärtsschritt sicherstellen zu können. Also kam uns die Idee, einfach eine neue *Unicorn*-Instanz zu erstellen und sie vom Anfang der Simulation bis zum gewünschten Punkt laufen zu lassen. Dies wäre nur dann nötig, wenn ein Nutzer vor den Schritt «Get Instruction» zurückspult (da in diesem Schritt ja die Werte aus Unicorn ausgelesen werden). Da Unicorn sehr schnell läuft, hatten wir bezüglich möglicher Performance-Einbussen keine Bedenken.

Alle sonstigen Felder von *Program* waren aber klonbar. Wir entwarfen ein neues Subset des Types *Program,* um die anderen Daten beibehalten zu können (siehe Abbildung 19).

ProgramTransactions enthält alle Felder des Interfaces *Program*, bis auf die *Unicorn*-Instanz «*ucInstance*» und die *disassemblerInstance* (Capstone). Die Disassembler-Instanz muss nicht erneut gespeichert werden, da diese statisch ist – sie wird lediglich zu Beginn einmal

```
interface ProgramTransactions {
  registersToShow: Array<RegisterID>;
  flagsToShow: Array<FlagID>;
  memoryAddress: number;
  memorySizeInBytes: number;
  codeAddress: number;
  codeSizeInBytes: number;
  code: number[];
}
```

Abbildung 19 - Interface 'ProgramTransactions'

initialisiert und danach werden keine weiteren Änderungen an ihr vorgenommen.

Zusammensetzung ucInstance

Die *ucInstance* wird mit einer Konfiguration und dem *Assembly*-Programmcode erstellt. Erstere ist immer dieselbe, Unterschiede ergeben sich nur beim übermittelten Programmcode.

Wir interagieren mit der Klasse *Unicorn* mittels der aktuellen Instruktion, die dann von Unicorn ausgeführt wird (Abbildung 20).

```
this.program.ucInstance.executeInstruction(this.state.currentInstruction);
```

Abbildung 20 - Interaktion mit Unicorn

Unsere *StateTransactions*-Datenstruktur speichert unter anderem alle Änderungen an *currentInstruction* ab. Jede Änderung entspricht dem Wert einer neuen Instruktion zum Zeitpunkt der Transaktion. Diese Logik machen wir uns zunutze, um die *ucInstance* neu zu bauen.

Sobald der Benutzer «Step Back» anklickt, wird die Methode «previousStep» aufgerufen. Entdeckt die Methode eine Neuzuweisung von *currentInstruction*, handelt es sich um einen «Get» Schritt und die *ucInstance* muss wiederhergestellt werden.

Wiederherstellung program

Um die Wiederherstellung durchzuführen sind folgende Schritte nötig:

- Wir erstellen ein neues *ucInstance*-Objekt, indem wir es mit der Standardkonfiguration und dem Assembly-Programmcode initialisieren.
- Wir führen jede in *StateTransactions* verzeichnete transaktionelle Änderung auf *currentInstruction* in umgekehrter Reihenfolge auf dem neuen *ucInstance*-Objekt aus. Dazu verwenden wir `ucInstance.executeInstruction`.
- Wir definieren eine neue *Program*-Instanz und setzen alle Felder dem dazugehörigen Eintrag in *ProgramTransactions* gleich. Zusätzlich hängen wir die aktuelle disassemblerInstance der neuen Program-Instanz an. Am Schluss wird die soeben neu konstruierte *ucInstance* ebenfalls angehängt.

Die Wiederherstellung des Zustandes ist damit vollendet.

Zusammenfügen aller Datenstrukturen (Transaction Store)

Um eine kohärente Übersicht zu gewährleisten und eine Zerfaserung zu verhindern, fügen wir alle Strukturen zusammen. Der *TransactionStore* ist ein Singleton und beinhaltet alle Änderungen am *State*, inklusive der Snapshots der *Program*-Instanz.

```
interface TransactionStore {
  stateTransactions: StateTransactions;
  programStates: [{version: Version; value: ProgramTransactions}];
}
```

Abbildung 21 - TransactionStore (finale Variante)

Nach einigen neu geschriebenen Testfällen sowie manuellem Überprüfen der States waren wir davon überzeugt, dass der Reverse Debugger nun zufriedenstellend funktionierte.

Auslesen der Transaktionen

Einführung

Jede gespeicherte Transaktion verfügt über eine *Version*. Schreitet der Simulator einen Schritt zurück, müssen die dazugehörigen Transaktionen evaluiert und möglicherweise geladen werden.

Der unten beschriebene Algorithmus ist in eine Schleife eingebunden, deren Abbruchbedingung dann eintritt, wenn zwischen dem letzten und aktuellen Durchlauf keine Veränderung festgestellt wurde.

Algorithmus

Für jede Property in *State* überprüfen wir den letzten Eintrag in *stateTransactions*. Er verfügt über zwei Bausteine: die Version, sowie den Wert.

Wir vergleichen die *Version* der aktuellen Transaktion mit der globalen Versionsverwaltung (inklusive der Reduktion des Schrittes).

Um zu entscheiden, wie mit der Veränderung weiter vorgegangen werden soll betrachten wir die beiden Fällen:

1. Version Master > Version Transaktion
2. Version Master <= Version Transaktion

Trifft Fall 1 zu, muss die Transaktion nicht weiterverarbeitet werden. Es handelt sich um eine in der Vergangenheit liegende Änderung, die nach wie vor aktuell ist, da es sich um das letzte Element in unserem Stack handelt. Der Stack bleibt unverändert, die Evaluierung ist abgeschlossen.

Handelt es sich um Fall 2, ist die Transaktion aktueller als die globale Version. Sie befindet sich für uns in der Zukunft, wird also erst bei einem nächsten Schritt wieder aktuell. Wir verwerfen den Eintrag von Stack, laden den nächsten Eintrag in den State und beginnen unseren Versionsvergleich erneut, mit einem weiteren Durchlauf des «Loops».

UI Anpassungen

Step Back Button

Das User Interface wurde neben den im Kapitel «Benutzeroberfläche» beschriebenen Änderungen mit einem zusätzlichen Element ausgestattet: dem «Step Back» Button. Er erlaubt dem Benutzer, den Reverse Debugger auszulösen, um einen Schritt zurück zu gehen.

Um dem Benutzer klarzumachen, dass nicht vor den Anfang des Programms hinaus gegangen werden kann, wollten wir den Button ausgrauen, sofern man sich im ersten Schritt der ersten Instruktion befindet.

Dank unserer globalen Versionsverwaltung war dies eine verhältnismässig schnelle Erweiterung (Abbildung 22).

```
public isInitialStep() {
  return this.versioning.step === 0 && this.versioning.nrOfInstructions === 1;
}
```

Abbildung 22 - Methode, um festzustellen, ob sich das Programm am Anfang befindet

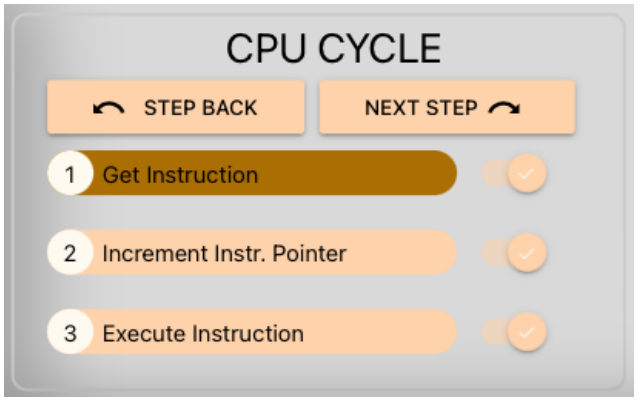

Abbildung 24 - Verhalten vor der Modifikation

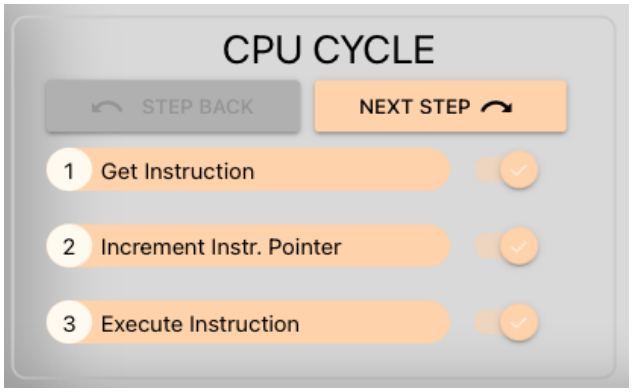

Abbildung 23 - Verhalten nach der Modifikation

Problematik UI-Updates (*byteInformation*)

In unserem ersten Prototypen, der in Ansatz 1 beschrieben wurde, trafen wir auf ein schwerwiegendes Problem. Das Aktualisieren des User-Interfaces war sehr zeitintensiv. Der Grund dafür war die Bindung der Daten in *state.byteInformation* an die *State*-Komponente.

Eine Zuweisung wie zum Beispiel

```
state.byteInformation = newByteInformation;
```

führte zu einer neuen Evaluierung aller *ByteVue.vue* Komponenten. Das gesamte Memory ist aus diesen aufgebaut. Jede dieser Bauteile enthält eigene Logik, um das CSS entsprechend einzufärben, Werte anzupassen etc.

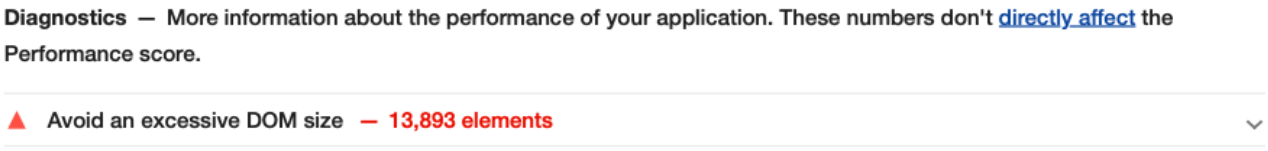

Abbildung 25 - Chrome Lighthouse Performance Benchmark (8.3.0)

Die Lösung für das Einfrieren und erneute Durchlaufen der gesamten Logik für alle 4’096 *ByteVue* Elemente war nicht offensichtlich.

Anstatt alle Objekte, die an *byteInformation* aufgehängt sind, neu zu bauen, beliessen wir die vorherige Instanz und wiesen nur die Subfelder neu zu (Abbildung 18)

```
const updatePointerInformation = (oldPointer: PointerInformation, newPointer:
PointerInformation) => {
      const updatedPointer = oldPointer;
      updatedPointer.pointerAddress = newPointer.pointerAddress;
      updatedPointer.pointerBytes = newPointer.pointerBytes;
};

updatePointerInformation(state.byteInformation.basePointerInformation,
newByteInformation.basePointerInformation);
```

Abbildung 26 - Neuzuweisung Subfelder

Die Lösung, wenn auch nicht besonders hübsch, funktioniert einwandfrei. Anstatt 2-3 Sekunden dauern die Updates nun unter 0.5 Sekunden.

In einer weiteren Iteration der Applikation wäre es jedoch sinnvoll, die *ByteVue.vue*-Komponente neu zu schreiben und dabei die Logik aus der Komponente auszulagern.

Fazit

Insgesamt sind wir äusserst zufrieden mit den Ergebnissen unserer Arbeit. Unser Ziel war es, den bereits sehr guten Simulator noch besser zu machen. Wir wollten dabei hauptsächlich Verbesserungen an der Bedienbarkeit vornehmen, sodass der Simulator ein praktisches Werkzeug zum Lernen wird. Dies ist uns in mehrerlei Hinsicht gelungen.

Erstens machen es unsere GUI-Verbesserungen angenehmer, das Programm zu verwenden. Dadurch kann die Zeit des Nutzenden primär für das Erwerben von Verständnis verwendet werden, anstatt auf die Bedienung des Programms verschwendet zu werden. Zweitens ist es uns gelungen, den Simulator erfolgreich um die Funktionalität des «Step Back» - Knopfes zu erweitern. Da man gerade am Anfang oft eine komplizierte Animation wiederholen möchte, ohne dabei das gesamte Programm erneut laufen zu lassen, dürfte auch dies viel zur einfacheren Nutzung des Simulators beitragen.

Auch aus technischer Perspektive waren wir erfolgreich: obwohl man jede beliebige Operation rückgängig machen kann, gibt es keine Performance-Einbussen und keine signifikante Erhöhung des benötigten Speichers. Der Simulator läuft also gleich schnell und effizient wie vorher. Wir haben einiges gelernt – sowohl über das bestehende Produkt als auch über den verwendeten Technologie-Stack. Es hat uns Freude bereitet, etwas Bestehendes weiterzuentwickeln und wir glauben, dass uns dies gut auf den späteren Berufsalltag vorbereitet.

Wir hoffen, dass der Simulator vielen zukünftigen Studenten helfen wird und freuen uns, eventuell während unserer Bachelorarbeit erneut daran arbeiten zu dürfen.

Literaturverzeichnis

[1] oftheheadland. "Colorblindly." https://chrome.google.com/webstore/detail/colorblindly/floniaahmccleoclneebhhmnjgdfijgg?hl=en (accessed.

[2] J. B. Engblom, "A review of reverse debugging," *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference,* pp. 1-6, 2012.

[3] A. Savidis and V. Tsiatsianas, "Implementation of Live Reverse Debugging in LLDB," *arXiv preprint arXiv:2105.12819,* 2021.

[4] P. Dovgalyuk, "What is reverse debugging? Classification of reverse debugging methods," *Journal of Physics Conference Series,* 2019.

[5] RogueWaveSoftware. "Finding hard-to-reproduce bugs with reverse debugging." https://totalview.io/sites/totalview/files/pdfs/white-paper-totalview-finding-bugs-reverse-debugging.pdf (accessed 13.12., 2021).

[6] Undo.io. "What is Reverse Debugging, and why do we need it?" https://undo.io/resources/reverse-debugging-whitepaper/ (accessed 13.12, 2021).

[7] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *J. of Math,* vol. 58, no. 345-363, p. 5, 1936.

[8] D. M. Clements. "Really Fast Deep Clone." https://github.com/davidmarkclements/rfdc (accessed 20.12., 2021).

[9] Mozilla. "Javascript Documentation - Proxy." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy (accessed 23.12., 2021).

[10] Mozilla. "Javascript Documentation - Reflect." https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Reflect (accessed 23.12., 2021).

Abbildungsverzeichnis