OST
Ostschweizer
Fachhochschule

Weiterentwicklung Prozessor-Simulator

Bachelorarbeit

Studiengang Informatik

OST - Ostschweizer Fachhochschule

Campus Rapperswil-Jona

Frühjahrssemester 2022

Autoren:	Michael Schneider, Tobias Petter
Betreuer:	Prof. Stefan Richter
Experte:	Dr. Ettore Ferranti
Gegenleser:	Dr. Thomas Bocek

OST
Ostschweizer
Fachhochschule

Inhaltsverzeichnis

Einleitung

Zu den Grundlagen der Ausbildung eines Informatikers gehört ein fundiertes Verständnis für die Funktionsweise von Computersystemen. Obwohl heutzutage zunehmend höhere Abstraktionsebenen wie beispielsweise Applikationsentwicklung in den Vordergrund treten, ist ein Verständnis der Hardware, die diese Applikationen ausführt, nach wie vor unerlässlich.

Im Zusammenspiel der Komponenten kommt dabei dem Prozessor (auch «CPU» genannt) die zentralste Rolle zu. Im Kern jedes Computersystems liegt der folgende Prozess: jede Zeile Code wird in vielen Zwischenschritten mehrfach verwandelt und endet als eine Reihe einfacher Instruktionen. Diese werden dann von der CPU ausgeführt.

Die OST hat diese Inhalte bisher im ersten Jahr des Informatikstudiums im Fach «Betriebssysteme» unterrichtet. Da neu angehende Studentinnen und Studenten oft Schwierigkeiten mit dem hohen Abstraktionsniveau des Stoffes hatten, wurde das Fach inzwischen ins zweite Jahr verlegt. Ausserdem wurde die Entwicklung eines graphischen Simulators vorgeschlagen, der den Studierenden auf ansehnliche Weise demonstrieren könnte, wie die Exekution von Assembly-Instruktionen in der CPU genau abläuft.

Eine erste Version dieses Simulators ist im Herbstsemester 2020 sowie im Frühjahrssemester 2021 von Eliane Schmidli und Yves Boillat im Rahmen ihrer Studien- respektive Bachelorarbeit entwickelt worden. Anschliessend haben die Autoren dieser Arbeit das Projekt übernommen und den Simulator während ihrer Studienarbeit im Herbstsemester 2021 weiterentwickelt.

Das Ziel der vorliegenden Arbeit war ein Abschluss dieser Weiterentwicklung und die vorläufige Fertigstellung des Projekts «Prozessor-Simulator». Die Applikation wurde grundlegend modernisiert und um eine Reihe von Features erweitert. Das Ziel der Arbeit war eine fehlerfreie, moderne Applikation, die Studierende und sonstige interessierte Personen (noch besser) beim Erarbeiten des Stoffs unterstützt.

Ausgangslage

Im Folgenden wird der Entwicklungsstand der Applikation zu Beginn dieser Arbeit beschrieben.

Der Simulator besteht aus zwei Teilen:

Zunächst sieht der Nutzende den Editor, in den Assembly-Code eingegeben werden kann. Die verwendete Sprache ist dabei NASM (Netwide Assembler [1]).

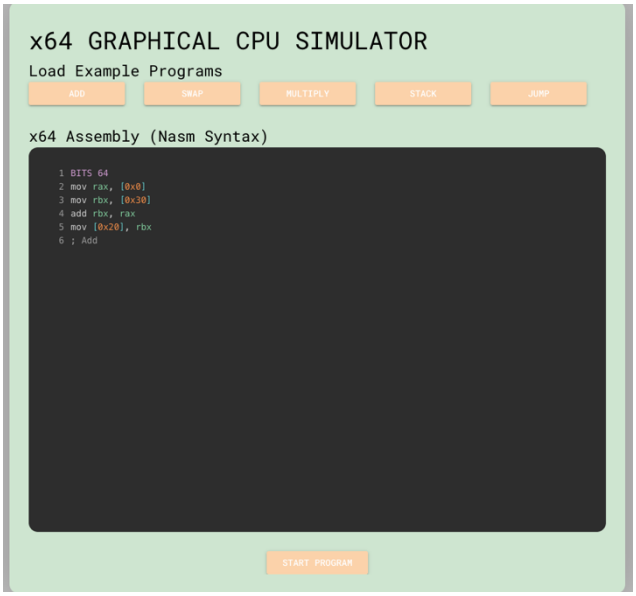

Abbildung 1 - Editor-Fenster

Der Editor enthält diverse Beispielprogramme, die durch Klicken geladen werden können. Diese dienen einem schnellen Einstieg in den Simulator - ohne dass sich Nutzende vorher selbst ein gültiges Assembly-Programm überlegen müssen.

Sobald die Eingabe des Programmcodes abgeschlossen ist, wird das Programm verarbeitet und in das Hauptfenster des Simulators geladen.

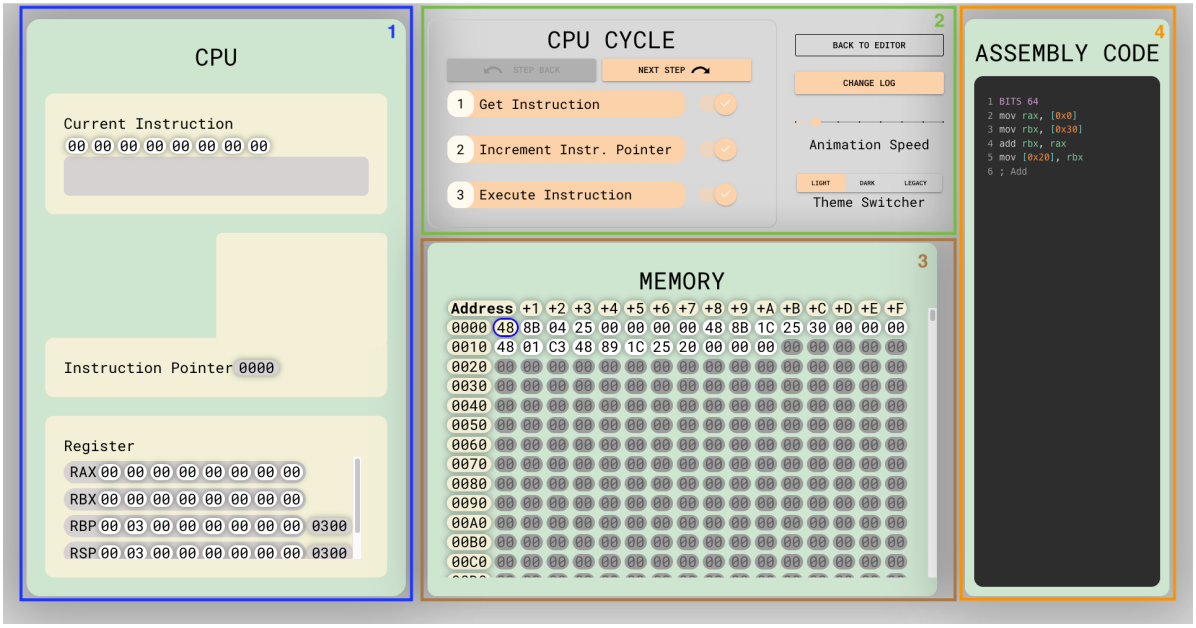

Abbildung 2 - Simulator-Hauptfenster zu Beginn der Arbeit

Dieses ist wie folgt unterteilt:

1) Prozessor mit aktuell geladener Instruktion, Instruktionszeiger sowie den momentan verwendeten Registern und Flags.
2) Kontrollbereich mit einer Darstellung des aktuellen Prozessorzyklus sowie den Vor- und Rückwärtsbuttons. Ausserdem lässt sich der Ablauf der Animationen mithilfe des Geschwindigkeitsreglers beschleunigen. Im Change Log werden vergangene Änderungen dargestellt. Der Theme Switcher erlaubt, verschiedene Farbschemen zu laden.
3) Inhalt des Arbeitsspeichers. Sichtbar sind die Instruktionen des aktuellen Programms, nicht verwendete Speicherbereiche sowie der Instruktionszeiger. Wer nach unten scrollt, sieht zusätzlich die Position des Stack Pointers und Stack Base Pointers.
4) Code Viewer zur Anzeige des vom Nutzenden eingegebenen Assembly-Codes.

Wird die Ausführung des Programms mittels Klicks auf den Button «Next Step» gestartet, schreitet der Simulator für jede ASM-Instruktion durch die drei angezeigten Schritte des Prozessorzyklus. Dabei werden alle Zustandsänderungen animiert, um die Aufmerksamkeit des Nutzenden auf die stattfindenden Veränderungen zu lenken.

Technischer Hintergrund & Architektur

Nach dem funktionalen Überblick wollen wir ebenfalls eine kurze Übersicht der Architektur geben. Da diese Arbeit ein bestehendes Projekt weiterentwickelt, wurde die Architektur in Gänze von unseren Vorgängern übernommen und durch uns kaum angepasst.

Der Simulator ist eine klassische Webapplikation, die aus der Kombination von **HTML**, **CSS** und **JavaScript** besteht. Konkret werden diese drei Grundkomponenten in der Applikation wie folgt erweitert:

- Statt reinem JavaScript ist **TypeScript** [2] im Einsatz, das Typensicherheit bietet
- Statt reinem HTML wird **Vue.js** [3] verwendet, ein Framework, das HTML in sogenannte Komponenten gliedert
- Die Benutzeroberfläche wird durch das Framework **Quasar** [4] definiert
- Zur Simulation des Prozessors wird der Emulator **Unicorn** [5] verwendet
- Zum Anzeigen der aktuellen Instruktion wird diese von den Disassemblern **Capstone** [6] und **Ndisasm** [7] rückübersetzt

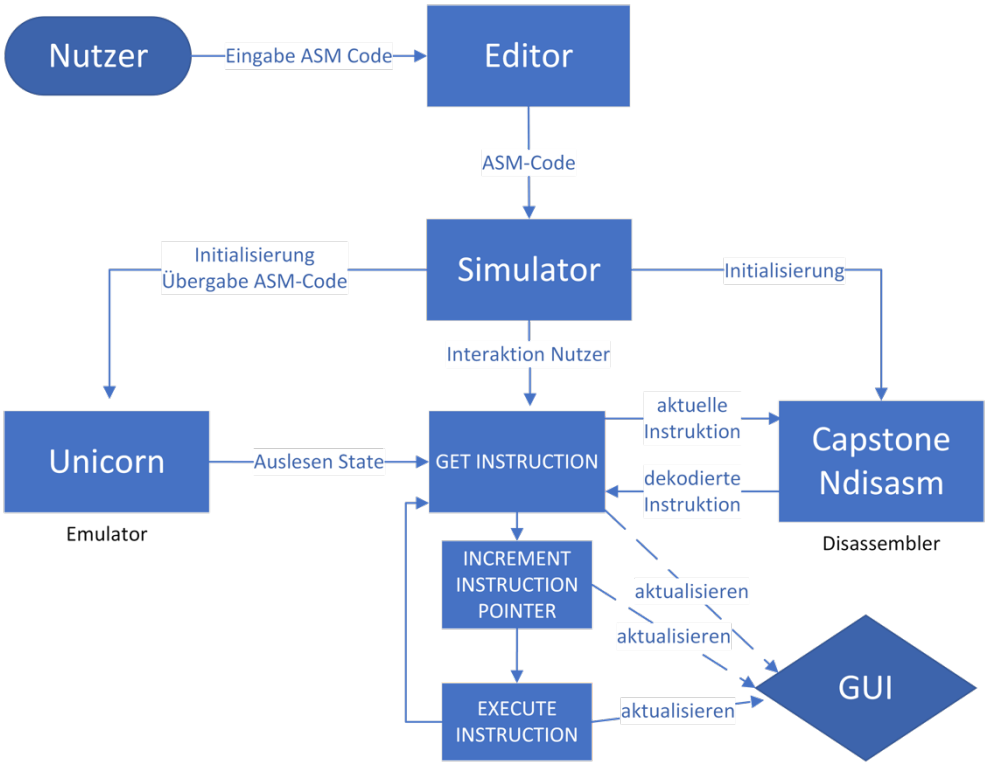

Abbildung 3 - Ablaufdiagramm des Simulators

Geplante Verbesserungen

Zu Beginn unserer Arbeit fanden mehrere Treffen mit Prof. Richter statt, der zugleich als Betreuer und «Kunde» fungierte. In diesen wurde besprochen, welche Erweiterungen an der Applikation vorgenommen werden könnten.

Dabei haben wir uns darauf geeinigt, zunächst den Code Viewer zu überarbeiten. Dieses Fenster wurde im Zuge unserer Studienarbeit (SA) als GUI-Verbesserung eingebaut, um den eingegeben ASM-Code jederzeit sichtbar zu machen. Das Ziel war damals, den Nutzenden klarzumachen, dass der von ihnen im Editor-Fenster eingegebene Code jetzt ausgeführt wird. Aus Zeitgründen wurden damals aber nur das simple Einblenden des Codes sowie die farbliche Hervorhebung der ASM-Syntax ermöglicht.

Wir haben drei neue Features identifiziert, die die Benutzererfahrung im Code Viewer unserer Meinung nach verbessern würden:

1. Wie im Debugging-Modus einer gewöhnlichen IDE sollte die **aktuell ausgeführte Instruktion farblich hinterlegt** werden, sodass Nutzende auf einen Blick nachvollziehen können, wo im Programm sie sich gerade befinden.
2. Durch Klicken auf eine Zeile Code sollte die **Ausführung des Programms zur entsprechenden Stelle im Code springen**. Dies würde Nutzern ersparen, sich durch längere Programme klicken zu müssen, nur um an der für sie interessanten Stelle anzukommen. Dabei soll auf eine beliebige Zeile (in der Vergangenheit oder Zukunft) geklickt werden können.
3. Ähnlich zu Feature 2) soll auch das **Setzen von Breakpoints** präzisere Navigation im eigenen Programm ermöglichen. Durch Klicken auf die Zeilennummer im Code Viewer sollten Breakpoints aktiviert sowie deaktiviert werden können. Dabei könnte dank unserem «Schritt-zurück» - Feature auch ein Breakpoint in der Vergangenheit gesetzt und zu diesem zurückgelaufen werden. Zwei zusätzliche Buttons sollen das **Vor- oder Rückwärtslaufen zu Breakpoints** (beziehungsweise dem Programm-Anfang/Ende) ermöglichen.

Alle drei dieser Features bedingen, dass unser Simulator eine Verknüpfung zwischen dem ASM-Code im Code-Viewer und den Instruktionen im Speicher herstellen kann. Ausserdem muss die aktuelle Position im Programm berücksichtigt werden. Zu Beginn der Arbeit waren die dargestellten Codezeilen nämlich lediglich normaler Text, ohne jegliche Verbindung zur Ausführung des Programms.

Die erste Herausforderung unserer Arbeit war es also, diese Verbindung herzustellen – sobald dieser Schritt gelöst worden wäre, könnten anschliessend die obengenannten Features problemlos implementiert werden.

Wir haben uns ausserdem eine Reihe von weiteren Verbesserungen für den Simulator überlegt:

4. Das Modifizieren der Breakpoint-Funktionalität, sodass **bedingte Breakpoints** möglich wären. Zwei Beispiele dafür:
   1) «halte an, wenn RAX = 2563»
   2) «halte an, wenn der Wert von RBX nicht mehr 0 ist»

   Diese Funktionalität würde neue GUI-Elemente erfordern, sodass die Bedingungen der Breakpoints definiert werde können.
5. Das **manuelle Editieren von Register- und/oder Speicherinhalten** während der Ausführung. Erlaubt Nutzenden, Teile ihres Programms dynamisch anzupassen. Man kann sich dadurch beispielsweise das Einfügen von Instruktionen sparen, die Speicher/Register mit Konstanten befüllen. Ausserdem erhält man die Flexibilität, während der Ausführung Werte anzupassen zu können, ohne dafür ein neues Programm schreiben und von vorne starten zu müssen.
6. **Hot Code Editing** – die Möglichkeit, das ausgeführte Programm zu modifizieren, während es ausgeführt wird. So könnten Nutzende bspw. spontan eine in der Zukunft liegende Instruktion anpassen, ohne dafür das gesamte Programm von vorne starten zu müssen.
7. **Post-Mortem Debugging** – die Möglichkeit zum Anzeigen oder Ausdrucken einer Übersicht am Schluss des Programms. Diese enthält alle Schritte des Programms und die daraus folgenden Änderungen an Speicher, Registern und Flags.

Bei der Priorisierung der Verbesserungen haben wir uns überlegt, welche davon am meisten Einfluss auf die Benutzererfahrung haben würden. Dabei sind uns die Veränderungen am Code Viewer am wichtigsten erschienen: sie würden den Simulator an herkömmliche IDEs angleichen, die Studierenden aus anderen Fächern bekannt sein könnten. Ausserdem greifen diese Features auch technisch ineinander, wodurch mehrere Verbesserungen mit weniger Aufwand implementiert werden können.

Punkte 5 und 6 haben wir als weniger wichtig priorisiert, da beide lediglich Abkürzungen für die Nutzenden ermöglichen und keine neue Funktionalität hinzufügen. Diese geringe Verbesserung im Verhältnis zu hohem erwartetem Aufwand erschien uns nicht vorteilhaft.

Daher priorisierten wir die Verbesserungen in der Reihenfolge der oben ersichtlichen Nummerierung. Wir entschieden uns, so viele davon zu implementieren, wie die Zeit zulassen würde. Mindestens wollten wir jedoch die Veränderungen am Code Viewer durchführen.

Neue Features

1. Hervorheben aktueller Zeile & Instruktions-Mapping

1.1. Motivation

Durch das in der Studienarbeit erfolgte Hinzufügen der Komponente „Code Viewer“ bekamen Nutzende die Möglichkeit, ihren Code als Referenz zu sehen, während er ausgeführt wird. Eines der primären Ziele dieser Arbeit war es, die Interaktion mit diesem Code zu ermöglichen. Dies sollte zu einem besseren Verständnis des Zusammenhangs zwischen geschriebenem Assembly-Code und ausgeführten Prozessor-Instruktionen beitragen.

Wir wollten damit beginnen, die in der Nutzeroberfläche präsentierten Informationen graphisch besser aufzubereiten. Als erstes zu implementierendes Feature wählten wir daher das farbliche Hervorheben der aktuell ausgeführten ASM-Codezeile. Diese Markierung verändert sich während der Exekution des Programms laufend. Dadurch sollte Nutzenden verständlich werden, wo in ihrem Code sie sich befinden und welche Instruktionen als nächstes ausgeführt werden. Ausserdem sollte dieses Verhalten den Nutzenden bekannt vorkommen: Debugger in konventionellen IDEs heben ebenfalls die aktuell ausgeführte Zeile hervor, sobald die Ausführung gestoppt wird. Wir gehen stark davon aus, dass Studenten im zweiten Jahr ihres Studiums bereits mit dieser Funktionalität vertraut sind. Dieser Wiedererkennungswert sollte sich daher positiv auf das Verständnis des Simulators auswirken.

1.2. Implementation Backend

Um die entsprechende Zeile im Code Viewer hervorheben zu können, muss das Backend die aktive Zeilennummer ans Frontend liefern. Das Problem dabei ist, dass das Backend nach der Initialisierung von Unicorn lediglich eine Reihe von Bytes im Speicher sieht und keinerlei Informationen über Instruktionen oder deren Zeilennummer besitzt.

Daher musste zuerst jeder Instruktion im Programm eine Positionsnummer zugewiesen werden. Anschliessend könnte man die Nummer der aktuell aktiven Instruktion auslesen und dem Code Viewer übergeben.

Wir wollten eine möglichst saubere, und zuverlässige Lösung, die in jedem Fall ein Mapping ermöglicht. Während einer Recherche zum Visual Studio

Code Debugger Protokoll [8] entdeckten wir das «DeZog»-Projekt [9], einen Emulator für viele alte Assembly-Sprachen. DeZog verfügt ebenfalls ein Mapping von Zeilennummer zu Instruktionen. Die Zeilennummern der Breakpoints werden in DeZog mit spezifischen Memorybereichen assoziiert [10]. Diese Methodik war einer der Inspirationsgründe für unseren Ansatz.

Da es von entscheidender Bedeutung war, die Performance der Applikation während der Ausführung von Programmen nicht weiter zu belasten, entschieden wir uns für eine JS-Map als Datenstruktur. Diese würde Instruktionen Zeilennummern zuweisen und könnte einmalig zu Beginn des Programms erstellt werden – während der Ausführung müsste sie dann lediglich ausgelesen werden.

Für die Erstellung dieser Struktur erstellten wir die Funktion «Map Instructions To Memory». Diese wird nach dem Einlesen des ASM-Codes und kurz vor der Übergabe der Kontrolle an den Nutzenden aufgerufen. Beginnend mit Adresse 0x0000 wird die Länge der ersten Instruktion mithilfe des Disassemblers ermittelt. Mit der erhaltenen Länge und der gegebenen Startposition (0x0000) lässt sich bei Unicorn auslesen, welche Instruktion sich an dieser Adresse befindet. Unser Simulator verwendet die zurückgegeben Daten und konstruiert ein Objekt mit dem Typ: «Instruction» daraus.

Aus der Instruktion, ihrem Start- und Endpunkt im Memory sowie einer fortlaufenden Zeilennummer erstellen wir ein neues «EditorLine» Objekt:

```
interface EditorLine {
  line: number;
  memoryAddressFrom: Address;
  memoryAddressUntil: Address;
  instruction: Instruction;
}
```

Dieses fügen wir anschliessend für schnelleren Zugriff in eine Map ein, die eine Zeile mit einer «EditorLine» verknüpft.

Nachdem alle relevanten Informationen für das erste «EditorLine»-Objekt gesammelt wurden, iterieren wir weiter bis:

```
Memoryposition 0x0000 + Länge der Instruktion
```

An dieser Stelle erstellen wir unser nächstes Objekt und fügen es zusammen mit der Zeilennummer 2 in die Map ein.

Anschliessend iteriert diese Funktion so lange, bis wir das Ende des allozierten Speichers erreicht haben. Das Resultat ist eine Datenstruktur, die alle Instruktionen und ihre Zeilennummern beinhaltet.

1.3. Implementation Frontend

Rein visuell war diese Veränderung relativ einfach zu implementieren. Der bislang verwendete Editor Prism.js [11] stiess mit dieser neuen Anforderung an seine Grenzen. Ausserdem würde man auch einige der anderen geplanten Features darin nicht implementieren können (z.B. Breakpoints). Daher ersetzten wir ihn durch eine HTML-Table. Der Assembly-Code (anfangs lediglich ein langer String) wird an den Zeilenumbrüchen aufgeteilt und zeilenweise in ein Array gefüllt, das dann den Inhalt der Table darstellt. Die Zeilennummern bilden eine eigene Spalte. Im Zuge dieser Überarbeitung passten wir den Code Viewer ausserdem farblich an unsere Farbschemata an und veränderten die Schriftart auf Prof. Richters Wunsch in eine Monospace-Schrift. Schliesslich bereiteten wir eine neue CSS-Klasse vor, die wir später der vom Backend gelieferten Zeile zuweisen wollten. Sie ändert die Hintergrund-Farbe, um die Zeile hervorzuheben. Ein Vorher-Nachher-Vergleich ist in **Abbildung 4** ersichtlich.

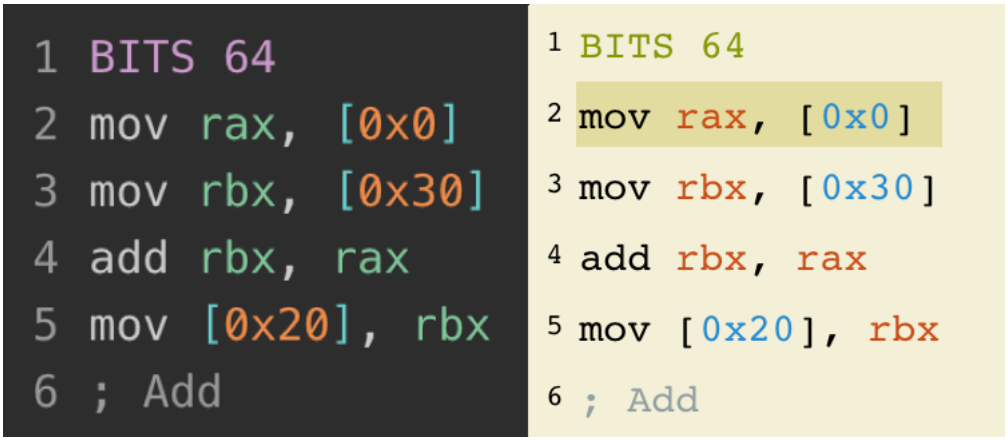

Abbildung 4 - Vergleich Hervorhebung aktive Zeile

Die Implementierung auf technischer Seite erwies sich wie erwartet als deutlich komplexer. Dank des bereits erwähnten Instruktions-Mappings konnte das Backend zwar die aktive Zeilennummer ans Frontend liefern, diese Information war allerdings vorerst unbrauchbar. Assembly-Programme enthalten nämlich Textzeilen, die nicht direkt in Instruktions-Bytes im Speicher umgewandelt werden. Es handelt sich dabei hauptsächlich um Pseudo-Instruktionen, Assembler-Direktiven sowie User-Kommentare.

Die Anweisung „BITS 64“ zu Beginn jedes unserer Programme beispielsweise sorgt dafür, dass NASM Instruktionen für einen 64-Bit-Prozessor generiert. Andere Beispiele sind das Setzen von Konstanten mit EQU, das Deklarieren von Daten mit DB / RESB oder das Verwenden von Kommentaren mit «;».

Diese Anweisungen werden von NASM zwar verarbeitet, dienen aber als Information an den Compiler und landen daher nicht als Bytes im Instruktionsspeicher. Es entsteht daher eine Diskrepanz zwischen der Anzahl Instruktionen im Backend und der im Frontend: das Backend «kennt» aufgrund der beschriebenen Instruction Mapping-Methode nur die Instruktionen, die tatsächlich im Speicher landen. Das Frontend jedoch zeigt *alle* Zeilen des Programms an, wodurch es insgesamt mehr (oder gleich viele) Zeilen beinhaltet.

Ein Beispiel ist in Abbildung 5 ersichtlich, in der rote Backend-Zeilennummern ins Beispielprogramm «Multiply» eingefügt wurden.

```
 1 BITS 64
 2 SECTION .text
 3 1 mov rax, 0x4E
 4 2 mov rbx, 0x3
 5   loop:
 6 3   add rcx, rax
 7 4   dec rbx
 8 5   cmp rbx, 0
 9 6   jne loop
10 7 mov [0x50], rcx
11 8 xor rax, rax
12 9 xor rcx, rcx
13 ; Multiply
```

Abbildung 5 - Vergleich Zeilennummern Frontend (grau) – Backend (rot)

Zeilen 1, 2, 5 und 13 werden nicht als Instruktionen auf den Speicher gemappt und sind daher für das Backend «unsichtbar». Dementsprechend ist auch die Zeilennummerierung unterschiedlich.

Daraus entstand die Notwendigkeit, ein Mapping zwischen den angezeigten Zeilen im Code Viewer und den vom Backend gemappten Instruktionen zu erstellen. Die vom Backend übermittelten Zeilennummern mussten vor ihrer Verwendung in Frontend-Zeilennummern «übersetzt» werden.
Wir entschlossen uns dazu, dies als eine JavaScript-Map namens *translationTable* zu modellieren. Für obengenanntes Beispiel würden wir also das folgende Mapping generieren:

Frontend-Zeilennummer	Backend-Zeilennummer
3	1
4	2
6	3
7	4
8	5

Frontend-Zeilennummer	Backend-Zeilennummer
9	6
10	7
11	8
12	9

Einträge für die Frontend-Zeilennummern 1, 2, 5 und 13 existieren nicht, da es dazu keine korrespondierenden Instruktionen im Backend gibt.

Diese Map wird einerseits mittels einfachem Lookup verwendet, um Frontend-Zeilennummern zu konvertieren und ans Backend zu liefern. Dies geschieht zum Beispiel, wenn der Nutzende auf eine Zeile klickt, um einen Breakpoint zu setzen. Andererseits kann auch ein Reverse Lookup für eine Konvertierung in die andere Richtung durchgeführt werden. Dies ist bei unserem aktuellen Problem notwendig, um die aktuell aktive Zeilennummer zu konvertieren, sodass die richtige Zeile hervorgehoben werden kann.

Für das Erstellen dieser Map benötigt man aber einen Weg, festzustellen, welche NASM-Zeilen als Instruktionen im Memory landen und welche nicht. Im **NASM Instruction Manual** [12] findet man in Kapiteln 3.2 («Pseudo-Instructions») und 7 («Assembler Directives») mehrere Listen dieser Pseudo-Instruktionen. Wir schrieben also einen simplen Check, ob diese Instruktionen am Anfang bzw. innerhalb von ASM-Zeilen stehen. Wenn ja, behandeln wir diese Zeilen als ungültig und ihnen wird keine Backend-Zeilennummer zugewiesen. Auf diese Weise lässt sich die *translationTable* einfach aufbauen. Es ist gut möglich, dass wir anhand des Manuals nicht abschliessend alle Instruktionen identifizieren konnten, die nicht auf Bytes im

Instruktionsspeicher mappen. Wir erachten jedoch die Wahrscheinlichkeit, dass Nutzende unseres Simulators dort komplexe Assembler-Direktiven anwenden, als eher gering. Die Zielgruppe besteht schliesslich hauptsächlich aus Studierenden am Beginn des Studiums.

2. Eingefärbte Instruktionen

2.1. Ausgangslage

Im Bereich «Memory» des Simulators werden nach Eingabe des Assembly-Programms alle Instruktionen als Bytes codiert abgelegt, beginnend bei Memory-Adresse 0x0000. Die Instruktionen sind dabei gemäss des Instruction Sets kodiert und können auch unterschiedliche Längen haben.

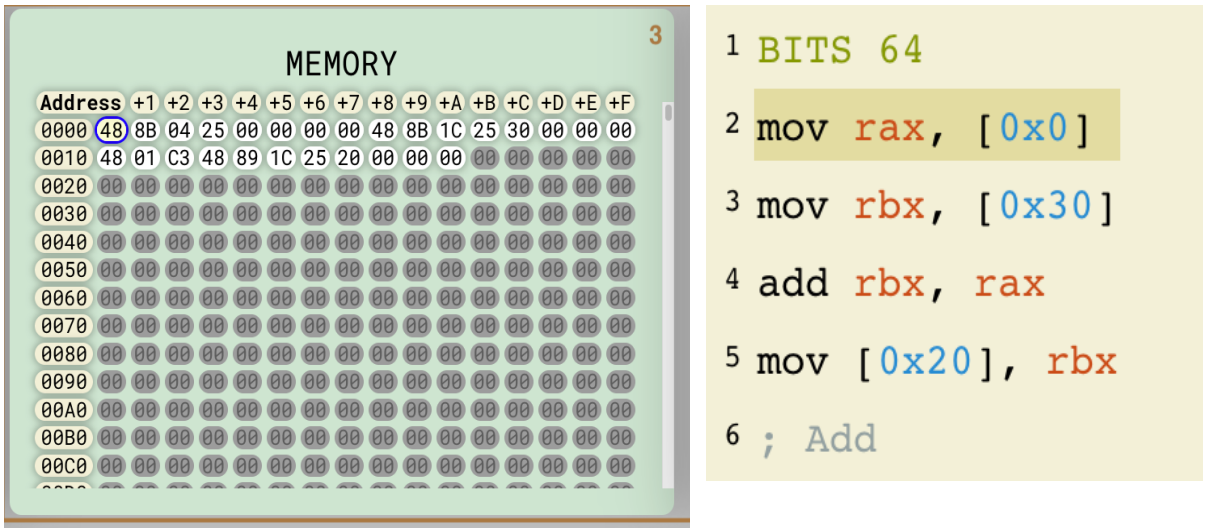

Abbildung 6 - Das Beispielprogramm «Add» und der entsprechend gefüllte Memory-Bereich

2.2. Motivation

In jedem «Get Instruction» - Schritt wird die nächste Instruktion in der Form von mehreren Bytes aus dem Speicher in den CPU-Bereich kopiert. Dieser Vorgang wird animiert.

Möglicherweise ist trotzdem nicht allen Nutzenden anfangs vollkommen klar, dass die Zahlen und Buchstaben im Speicher ihre codierten ASM-Instruktionen darstellen. Im Zuge unseres Vorhabens, dargestellte Informationen zugänglicher zu machen, kam unserem Betreuer Prof. Richter während eines Fortschrittmeetings eine Idee: er schlug vor, die Instruktionen im Speicher alternierend einzufärben, damit man einzelne Instruktionen auch

in ihrer codierten Form voneinander unterscheiden kann. Dies würde Nutzenden dabei helfen, ihre Instruktionen im Speicher wiederzufinden. Ebenfalls hilft es zu verstehen, wie der Instruction Pointer über den Memorybereich wandert und dass Instruktionen unterschiedliche Längen haben können.

2.3. Technische Umsetzung

Die Voraussetzung für das Einfärben der Instruktionen wurde mit der Funktion *Map Lines To Memory* (siehe Abschnitt **1.2**) bereits geschaffen. Die dort erstellte Datenstruktur ermöglicht uns einfachen Zugriff auf alle Instruktionen im Speicher und enthält Informationen über Beginn und Ende ihres Speicherbereichs.

Somit blieb im Backend nur mehr zu ermitteln, ob ein Byte zu einer geraden oder ungeraden Instruktion gehört. Dies wird vom «colorInstructionsService» gemacht, der die zwei neuen Arrays

- evenInstructionBytes
- unevenInstructionBytes

auf «State.byteInformation» mit den IDs der jeweiligen Bytes befüllt.

Aufseiten des Frontends bestehen die Bytes im Speicher aus den «Byte.vue»-Komponenten. In diesen wird unter anderem definiert, wie sie für das GUI dargestellt werden soll. Den gewünschten Farb-Effekt erreichen wir mit zwei CSS-Klassen, die das Byte in zwei unterschiedliche Farben einfärben:

Abbildung 7 - ungerades Byte in blauer Farbe

Abbildung 8 - gerades Byte in oranger Farbe

Sofern die «id» des Bytes also in einem der obengenannten Arrays vorkommt, wird eine CSS-Klasse mit der jeweiligen Hintergrundfarbe zugewiesen und das Byte damit richtig eingefärbt.

2.4. Ergebnis

Das finale Produkt sieht wie folgt aus:

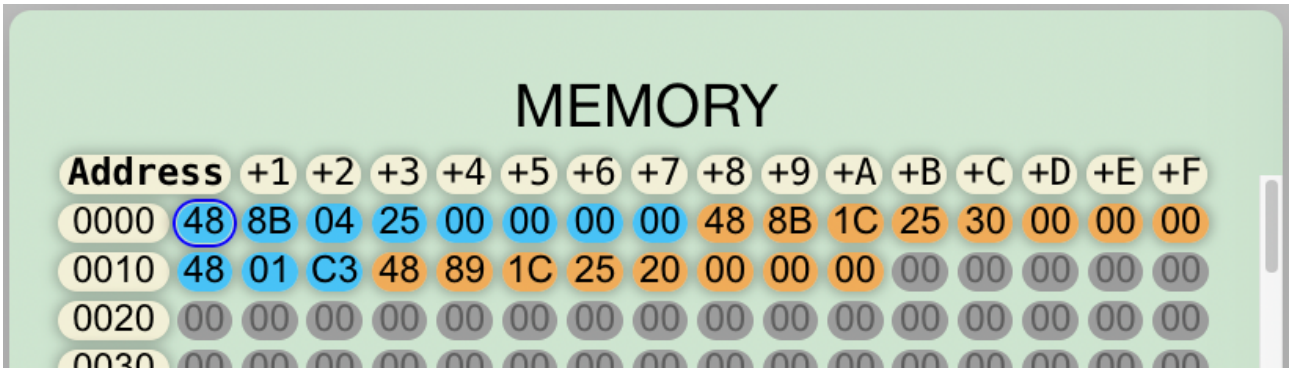

Abbildung 9 - Der Speicherbereich mit alternierend eingefärbten Instruktionen

3. Breakpoints

3.1. Motivation

Programmieren geht Hand in Hand mit Debuggen. Es gibt wohl kaum Programmierer, die noch nie einen Debugger verwendet haben. Zum Feststellen von Fehlverhalten im eigenen Code ist ein solches Werkzeug unabdingbar. Man könnte argumentieren, das Debugging sei der wissenschaftlichste Teil im Leben eines Informatikers: das Verhalten eines Prozesses (hier: Programmes) wird beobachtet und daraus werden Schlüsse über seine Funktionsweise gezogen. Der Beobachter lernt dadurch viel über den Prozess und seine Eigenschaften.

Dieser Aspekt ist der Hauptgrund, wieso wir Breakpoints in den Simulator einbauen wollten. Sie erlauben Nutzenden, den Prozessor noch genauer bei der Arbeit zu beobachten, indem sie an präzise ausgewählte Stellen in ihrem ASM-Programm springen können. Die daraus gezogenen Schlüsse erlauben ihnen, das Objekt der Untersuchung besser zu verstehen.

3.2. Implementation Backend

Die Umsetzung der kompletten Funktionalität erforderte einige Vorarbeit. Wir wollten mit einem Bottom-Up-Approach an das Problem herangehen und haben deshalb mit der Datenstruktur Breakpoint begonnen.

Definition eines Breakpoint-Typs

Was genau ist ein Breakpoint? Bevor wir einfach begonnen haben, ein solches Objekt selbst zu implementieren, warfen wir zur Orientierung einen Blick darauf, wie andere Entwicklungsumgebungen damit umgehen.

Im weltweit meistgenutzten Open-Source Code-Editor **VSCode** [13] setzt sich ein Breakpoint beispielsweise aus diesen Feldern zusammen [14] :

```
Id: string
Enabled: boolean
Condition: string
hitCondition: string
logMessage: string
```

Die *Id* sorgt für eine eindeutige Identifizierung des Breakpoints und wird nie mehrfach vergeben. *Enabled* zeigt an, ob der Breakpoint aktiviert ist oder nicht. Die *Condition* spezifiziert die Bedingung, unter der der ausgeführte Programmcode angehalten werden soll (nur bei bedingten Breakpoints relevant). Die *Hit Condition* erlaubt es, eine bestimmte Anzahl von Breakpoint-Auslösungen zu ignorieren und erst bei beispielsweise dem fünften Passieren des Breakpoints die Exekution anzuhalten. Die *logMessage* ist ein Platzhalter für allfälligen Text, der dem Nutzenden dargestellt werden soll.

Für unseren benötigten Funktionsumfang würden wir gerne in der Lage sein, eine Zeilennummer auf dem Breakpoint zu speichern, damit wir ihn einem der in Abschnitt **1.2** beschriebenen Instruction-Objekte zuweisen können. Ausserdem soll es (wie in VSCode) möglich sein, eine Bedingung (Condition) für das Auslösen des Breakpoints anzugeben. Wir haben die Struktur unserer Breakpoint-Objekte also wie folgt definiert:

```
interface Breakpoint {
  line: number;
  condition?: Condition;
}
```

Das Fragezeichen nach der Condition signalisiert Typescript, dass die Bedingung optional ist.

Breakpoint-Sammlung

Mit dem definierten Typ sind wir nur in der Lage, Breakpoints zu erstellen. Als nächstes gilt es, diese auch abspeichern zu können. Sobald ein neuer Breakpoint erstellt wurde, wird er an die Funktion *setBreakpoint* übergeben. Diese Methode speichert den Breakpoint in einem **Array** ab.

Die Verwendung der Datenstruktur Array besitzt dabei Vor- und Nachteile. In einem ersten Versuch wurde ein JavaScript-**Set** [15] gewählt. Der Vorteil an einem Set ist die Eigenschaft, dass Einträge einzigartig sind, wir müssen also nicht überprüfen, ob ein Breakpoint mehrmals gesetzt wurde. Der grösste Nachteil ist, dass keine Möglichkeit besteht, die Einträge eines Sets zu sortieren. Die Datenstruktur fügt die Einträge in der Reihenfolge ein, in der sie hinzugefügt wurden. Für uns ist das problematisch, da wir eine geordnete Datenstruktur benötigen: wir wollen in der Lage sein, den nächsten oder letzten Breakpoint ausgehend von der aktuellen Position im Code ohne grossen Ressourcenaufwand ausfindig zu machen. Um die Daten zu sortieren, ist bei einem Set jedes Mal eine Typenkonversion in ein Array nötig, worauf dieses sortiert werden muss. Wegen dieser uneleganten und nicht performanten Lösung haben wir uns für ein (sortiertes) Array entschieden.

Um eine ähnliche Funktionalität wie bei einem Set zu erreichen, überprüfen wir beim Einfügen eines neuen Eintrags, ob bereits ein Breakpoint auf der korrespondierenden Zeile gesetzt wurde. Falls nicht, wird er normal eingefügt. Falls ja, passiert bei normalen Breakpoints nichts. Bei bedingten Breakpoints wird jedoch die Bedingung ersetzt. Das Löschen von Breakpoints wird via Angabe der Zeilennummer ermöglicht.

Kontinuierliche Ausführung

Eine der Anforderungen an unsere Arbeit war, dass der Simulator «kontinuierlich oder schrittweise» laufen könnte. Auch die meisten anderen Debugger ermöglichen einen Vorlauf, der gegebenenfalls durch Breakpoints unterbrochen wird, ansonsten aber das Programm bis zu seinem Ende ausführt. Bei uns wird dies durch «nach hinten laufen» ergänzt, da unser Debugger auch in der Ausführung zurück gehen kann.

Da geplant war, den Nutzenden zwei neue Buttons zur Verfügung zu stellen («Run Forward», «Run Back»), mussten wir dieses Verhalten nun also implementieren. Grundsätzlich findet beim Klick eines der beiden «Run»-Buttons ein Methodenaufruf statt: mithilfe der aktuellen Instruktionsnummer wird im Breakpoint-Array überprüft, ob vor (bzw. nach, je nach Button) der

aktuellen Instruktion noch ein Breakpoint existiert. Falls ja, läuft die Ausführung bis dorthin. Falls nein, läuft das Programm bis zum Anfang/Ende durch.

Um herauszufinden, ob wir uns am Anfang des Programms befinden, war unsere eigene Vorarbeit in der Studienarbeit sehr hilfreich. Teil der Reverse Debugging-Komponente war ein Versionierungssystem, das jedem Schritt ein Objekt zuweist.

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

Im Zusammenhang mit dieser Versionierung hatten wir ausserdem eine Methode «isInitialStep» geschrieben. Diese überprüft, ob die Zahl der Instruktion (*nrOfInstruction*) im aktuellen Schritt 1 ist und teilt uns so mit, ob wir den ersten Schritt (den Start des Assembly-Programms) erreicht haben.

Ist also kein Breakpoint vor der aktuellen Position vorhanden, laufen wir so lange zurück, bis *isInitialStep* **true** zurückgibt.

Analog zum vorhergehenden Verhalten muss es auch möglich sein, ans Ende des Programmes zu gelangen. Auch hier haben wir im «StepController» eine Methode geschrieben - «isLastStep».

Ein alter Bug

Uns ist allerdings beim Verwenden dieser Methode fehlerhaftes Verhalten der Applikation aufgefallen. Der Simulator speichert den in Assembly-Instruktionen konvertierten Bytecode als Uint8-Array ab. Die Länge dieses Arrays wurde bislang verwendet, um die Länge des Programmcodes zu ermitteln und somit auch den Endpunkt des Programms im Speicher festzustellen. Dies funktioniert meistens, wird dann aber zum Problem, wenn im ASM-Code Daten initialisiert werden. Beispielsweise kann mit den ASM-Instruktionen

```
var:
  dq 0x1337
```

ein Quad-Word (= 8 Bytes) im Speicher zugewiesen und der Variable «var» zugeordnet werden.

Der Unicorn-Emulator führt diese Zuweisung im Speicher direkt nach der letzten Instruktion des Programms durch. Dadurch vergrössert sich der als Instruktionsspeicher definierte Speicherbereich, ohne dass tatsächlich eine Instruktion dazukommt.
Dies führt bei der oben beschriebenen Ermittlung der Länge des Instruktionsspeichers zu einem zu grossen Wert. In der Applikation unserer Vorgänger äussert sich dies darin, dass das Ende des von ihnen geschriebenen SWAP-Beispielprogramms nicht ordentlich abläuft. Der Simulator läuft über das Ende der Instruktionen hinaus und interpretiert den Inhalt der initialisierten Daten als Instruktionen.

Wir stiessen auf dieses Problem, weil unsere Funktion *MapLinesToMemory* deshalb nicht korrekt funktionierte und sich das SWAP-Programm nicht mehr starten liess.

Um diesen Bug zu fixen, haben wir deshalb *MapLinesToMemory* erweitert. Wir verwenden die Länge des Uint8-Arrays nicht mehr, sondern iterieren mit der in Abschnitt **1.2** beschriebenen Methode über den Instruktionsspeicher. Sobald der Disassembler keine gültige Instruktion mehr erkennt, markieren wir dies als das Ende des Instruktions-Speicherbereichs.

Weiterentwicklungspotential: Unsere Lösung, um die Länge zu ermitteln funktioniert bei all unseren Testprogrammen und allen Testfällen. Allerdings führt sie zu Problemen, wenn deklarierte Variablen zufälligerweise einem gültigen Op-Code gleichen. Sie werden dann als Instruktion erkannt und vom Simulator fälschlicherweise als solche markiert.

Eine korrekte Erkennung des Programmendes ist vermutlich nicht trivial, da man über mehr Informationen verfügen muss, als im Uint8-Array vorliegen. Möglicherweise bietet Unicorn diese Information an. Wenn nicht, müsste vermutlich der Assembly-Code analysiert und die Länge der Instruktionen daraus errechnet werden. Da NASM eine Architektur ohne fixe Instruktionslänge ist, dürfte diese letzte Variable eher komplex zu implementieren sein.

Leider entdeckten wir die Unstimmigkeit mit «SWAP» und der Codelänge erst relativ spät – in einer Phase, in der wir mit dem Programmieren fast fertig waren. Dadurch blieb keine Zeit, dieses Problem selbst zu lösen. Wir empfehlen dies jedoch für mögliche Folgearbeiten.

Mit der neu errechneten Länge kann «isLastStep» nun mithilfe des Instruction Pointers (meistens) herausfinden, ob wir am Ende des Programms angekommen sind.

Wir verwenden «Next Step» beim Vorwärtslauf also so lange, bis «isLastStep» **true** zurückgibt.

3.3. Implementation Frontend

Zieldesign

Das gewünschte Design für Breakpoints in der Benutzeroberfläche war eigentlich von Beginn an recht klar – wir wollten uns an herkömmlichen Debuggern in viel verwendeten IDEs orientieren, um unsere Nutzenden nicht unnötig zu verwirren. Ein solches Design erlaubt das Klicken auf oder neben Zeilennummern, um auf der entsprechenden Zeile einen Breakpoint zu setzen. Meist wird ein solcher anschliessend durch einen roten Punkt markiert. Beispiele dafür sind in Abbildung 10 ersichtlich.

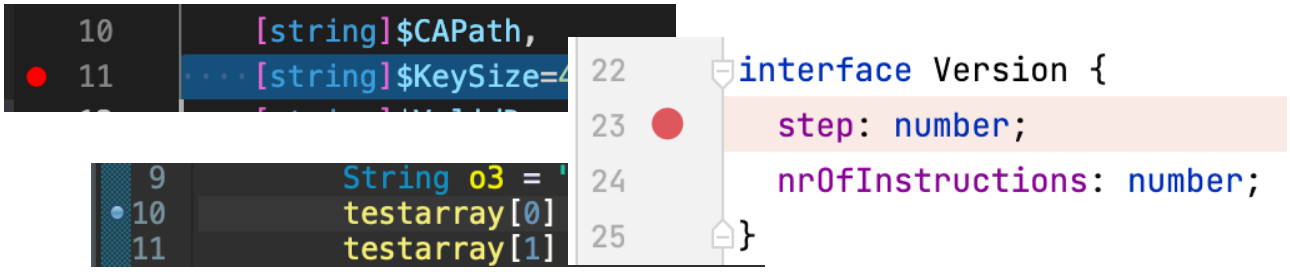

Abbildung 10 - Breakpoints in Visual Studio Code (oben links), Webstorm (rechts) und Eclipse

Die in Abschnitt **1.3** eingeführte HTML-Table im Code Viewer musste nun also erweitert werden. Aus Platzgründen entschieden wir, das rote Breakpoint-Symbol anstatt wie bei den oben gezeigten IDEs neben den Zeilennummern direkt unter ihnen zu platzieren. Für bessere Lesbarkeit werden die Zahlen dafür fett. Das Frontend ruft beim Klick auf eine Zeilennummer die Methode «toggleBreakpoint» auf, wodurch ein neuer Breakpoint hinzugefügt oder ein bereits gesetzter Breakpoint wieder entfernt wird.

Verhalten

An dieser Stelle bemerkten wir allerdings ein kleines Problem: was, wenn der Nutzende auf eine Zeile klickt, die «ungültig», also im Backend nicht gemappt ist (zum Beispiel ein Kommentar)? In anderen IDEs passiert in so einem Fall einfach nichts. Wir benötigen die Information über die Gültigkeit

einer Zeile bereits beim Aufbau der *translationTable* und speichern sie dann als eine Map namens *nextValidElementTable*. Daher wäre diese Implementation auch für uns einfach möglich gewesen. Allerdings wollten wir noch einen Schritt weiter gehen und unseren Nutzerinnen und Nutzern sozusagen unter die Arme greifen, indem wir an der nächstmöglichen Stelle einen Breakpoint setzen. Dank der obengenannten *nextValidElementTable* erforderte dies nur einen einfachen Lookup.

Wir demonstrierten Prof. Richter diese Funktion. Er wies uns darauf hin, dass dieses Verhalten bei längeren Segmenten «ungültigen» Codes (beispielsweise eine Reihe von Variablendeklarationen) verwirrend auf den Nutzenden wirken könnte. Beim Klick auf eine Zeilennummer würde plötzlich viel weiter unten im Programm ein Breakpoint gesetzt. Wir befürchteten, dadurch unsere Benutzeroberfläche weniger intuitiv gestaltet zu haben. Dies könnte zu fehlerhafter Bedienung des Simulators führen. Daher beschlossen wir, die Neuerung etwas einzuschränken: nur wenn direkt unterhalb einer ungültigen Zeile eine gültige liegt, wird dort ein Breakpoint gesetzt. Andernfalls passiert nichts.

Neue Buttons

Zu guter Letzt fehlten nur noch die zwei in Abschnitt **3.2** unter *Kontinuierliche Ausführung* erwähnten neuen Buttons. Sie sollten den Nutzenden die Möglichkeit bieten, die Applikation kontinuierlich vorwärts oder rückwärts laufen zu lassen. Wir fügten die Buttons direkt über den bestehenden Schaltflächen «Step Back» und «Next Step» ein. Für die Icons nutzten wir Symbole aus der Google Material Symbols Library [16] und ersetzten bei der Gelegenheit zur Vereinheitlichung auch gleich die von unseren Vorgängern erstellten, selbst definierten SVGs durch Icons aus der Library.

Zu Beginn unserer Studienarbeit existierte lediglich der Button «Next Step» in einer eigenen Vue-Komponente. Um «Step Back» einzufügen, kopierten wir einfach die bestehende Komponente und passten sie an, da eine Abstraktion für nur zwei Buttons unnötig erschien. Da nun aber vier Buttons vorhanden waren, lohnte sich die Auslagerung in eine eigene Komponente *ControlButton.* Somit muss von der übergeordneten Komponente *CpuCycle* lediglich ein Array mit Buttons erstellt und mit den gewünschten Werten (Button-Text, Tooltip-Text, Icon) befüllt werden. Via der Vue-Funktion **v-for** kann dann über das Array iteriert werden und alle Buttons werden wie gewünscht erstellt. Dabei wird in zwei Kategorien «backButtons» und «forwardButtons» unterschieden, da letztere am Ende des Programms ihre

Funktionalität anpassen und zu «Restart Program» werden. Am Ende sah das neue Design so aus:

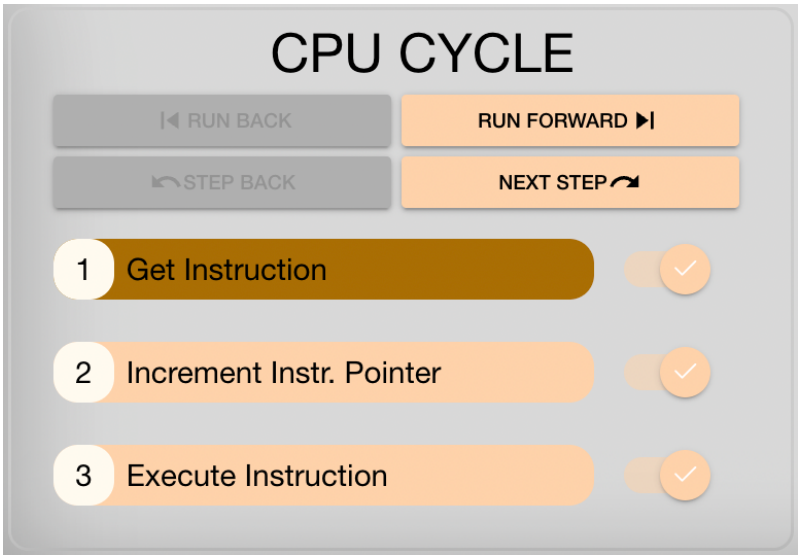

Abbildung 11 - CPU Cycle Komponente

Beim Klick auf «Run Forward» läuft der Simulator automatisch und ohne Animationen durch alle Instruktionen, bis entweder ein Breakpoint, bedingter Breakpoint (siehe nächstes Kapitel) oder Watchpoint (siehe Kapitel **5**) auslöst oder das Ende des Programmes erreicht ist. «Run Back» funktioniert analog, läuft aber rückwärts an den Anfang des Programms.

4. Bedingte Breakpoints

4.1. Motivation

Die Grundlage für reguläre Breakpoints war gesetzt. Wir waren nun in der Lage, Breakpoints zu setzen und zu löschen und den Programmfluss an der passenden Stelle anzuhalten. Normale Breakpoints haben allerdings einige Einschränkungen, die ihre Verwendung mitunter mühsam machen kann. Läuft das Programm beispielsweise in einem Loop und erst die fünfundzwanzigste Ausführung ist für das Debugging interessant, so muss ein gesetzter Breakpoint möglicherweise die ersten vierundzwanzig Durchläufe lang immer wieder manuell übersprungen werden. Dies ist sowohl mühsam als auch zeitaufwändig. Hier wäre ein bedingter Breakpoint, der beispielsweise die Loop-Variable *i* mittels Bedingung «i = 24» überprüft, ideal.

In unserem nächsten Schritt wollten wir es den Nutzenden daher ermöglichen, Bedingungen zu konstruieren, die definieren, wann das Programm an einem gegebenen Breakpoint halten soll. Wichtig ist hierbei die Abgrenzung zu *Watchpoints*, die zeilenunabhängig definiert werden und nach jedem Schritt automatisch überprüft werden.

Aufgrund diverser Gründe, die in Abschnitt **4.3** besser beschrieben werden, entschieden wir uns dafür, die Bedingung mithilfe eines Strings festzulegen.

4.2. Implementation Backend

Ein String (= Kette von Zeichen) muss für eine Auswertung interpretiert werden. Diese Interpretation wird auch *Parsing* genannt. Einen komplett eigenen Parser zu schreiben, hätte den Rahmen dieser Bachelorarbeit gesprengt. Deshalb machten wir uns auf die Suche nach einer geeigneten Hilfssoftware - einer Bibliothek, auf der wir aufbauen konnten.

Fortgeschrittenes Parsing kann über sogenannte Abstract Syntax Trees [17] abgewickelt werden. Mit dem Projekt «AST Explorer» [18] können solche Syntax Trees in JavaScript untersucht werden. Als Beispiel können wir die folgende Zuweisung inspizieren:

```
let expression = ["RAX = 0", "RBX = 8"];
```

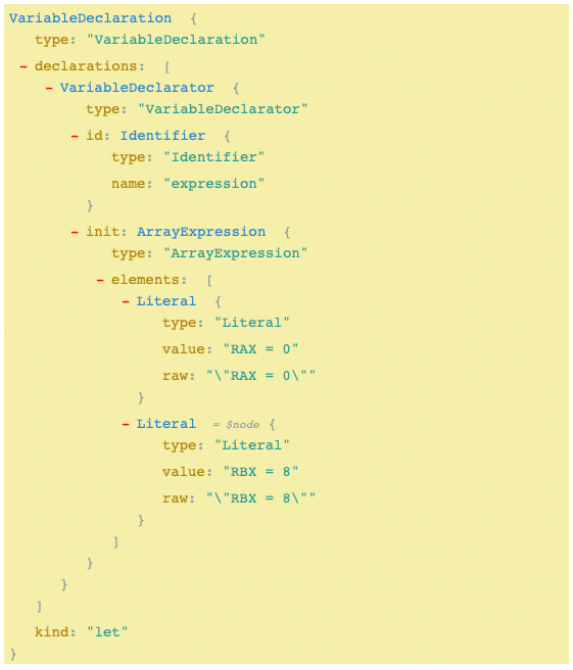

Abbildung 12 - Abstract Syntax Tree - Ausschnitt der Deklaration

Wir können über diese Bäume iterieren und die von uns benötigten Felder oder auch Blätter auslesen und verwerten. Die Verwendung von ASTs hätte vermutlich gut funktioniert.

Allerdings fanden wir während unseren Recherchen über Abstract Syntax Trees ein Package namens «Expression Parser» [19]. In diesem kann eine Sprache definiert werden, nach deren Regeln der eingegebene String ausgewertet werden soll. Diese Sprache wird auch *Formula* genannt. Es ist möglich, darauf Operationen und auch Operatoren zu definieren. Für genauere Dokumentation empfehlen wir die GitHub-Seite [20] des Projekts.

Dieses Package verfügt über alle Eigenschaften, die wir benötigen, um unseren Parser zu schreiben. Da es gleichzeitig wesentlich einfacher zu verwenden war als ASTs, beschlossen wir, es an ihrer Stelle einzusetzen.

Einleitung

Bedingungen bestehen aus Operanden und Operationen. Als Beispiel betrachten wir:

```
RAX = 488B042500000000
```

Hier wird das CPU-Register **RAX** (Operand 1) mit der hexadezimalen Adresse 488B042500000000 (Operand 2) verglichen (Operation). Sofern **RAX** diesem Wert entspricht, muss der Parser den Wert **true** zurückgeben.

Der Parser trennt die drei Bestandteile der Bedingung wie folgt auf:

Abbildung 13 - Aufteilung einer Bedingung

Die Herausforderung hier hängt mit dem ersten Operanden zusammen - der Parser weiss von sich aus nicht, was der Wert „RAX“ zu bedeuten hat. Wir mussten also eine Methode schreiben, die **RAX** sowie alle weiteren Register und Flags ausliest und einen Hex-Wert zurückgibt, sodass der Parser mit diesen arbeiten kann.

Operanden

Die Library bietet uns die Möglichkeit, jeden Term vor der Verarbeitung durch eine Funktion zu modifizieren. Dies Funktion müsste also in diesem Fall eine

Übersetzung der Zeichenkette «RAX» zum tatsächlichen Register-Objekt **RAX** erstellen, dessen Wert auslesen und dem Parser zurückgeben.

Als Grundlage für diese Übersetzung haben wir alle durch Unicorn erfassten Register und Flags ausgelesen und in einem Enum abgespeichert. Aus Gründen der Lesbarkeit sprechen wir im Rest des Kapitels lediglich von Registern – Flags sind aber immer ebenfalls gemeint.

```
export enum RegisterOperand {
  …
  RAX = 'RAX',
  RBP = 'RBP',
  RBX = 'RBX',
  …
}
```

Wir verfügen jetzt also über das Wissen, welche Register existieren.

Die vom Parser bereitgestellte Methode muss jetzt also prüfen, ob die Zeichenkette «RAX» in der Datenstruktur vorhanden ist. Ist diese vorhanden, handelt es sich um ein korrektes Register. Als nächstes soll dieses Register ausgelesen werden.

Die Register und ihr Inhalt sind auf dem Objekt *State* abgelegt. Es beinhaltet den aktuellen Zustand des Simulators in der Form, die den Nutzenden präsentiert wird. Wir vergleichen nun also das vorher ermittelte Register-Objekt mit den auf dem *State* vorhandenen Registern. Ist es vorhanden, wird der aktuelle hexadezimale Wert zurückgegeben. Dieser Wert wird dann an den Parser zurückgegeben und ersetzt dort den Register-Namen:

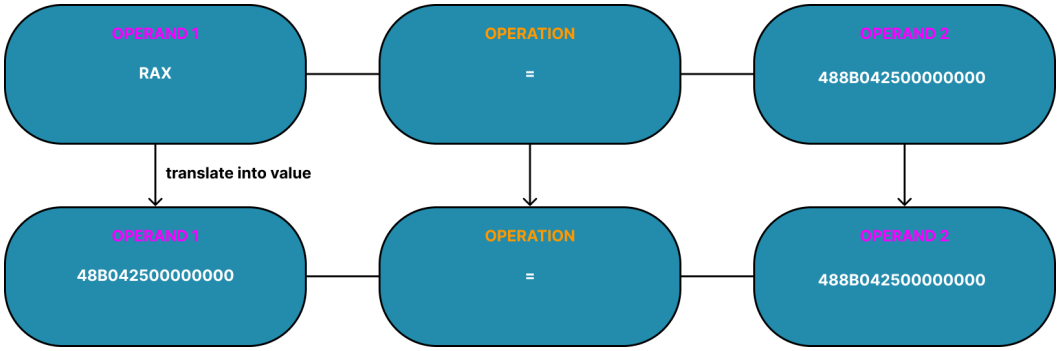

Abbildung 14 - Translation von Operand 1 in einen hexadezimalen Wert

Dieser Schritt wird für alle Operanden in der Bedingung vorgenommen. Danach wird die Bedingung mittels der Operation aufgelöst und ein Boolescher Wert **true** oder **false** zurückgegeben.

Weitere Operationen

Bis anhin haben wir die Aufgabe betrachtet, ein Register mit einem numerischen Wert zu vergleichen. Zusätzlich zum Vergleichen sollen auch weitere Rechenoperationen möglich sein. Wir haben die gängigen Infix-Operationen in den Parser integriert:

Operation	**Erklärung**	**Gleichheits-Operation**	**Erklärung**
>	Grösser	<	Kleiner
+	Addition	>=	Grösser oder gleichwertig
-	Subtraktion	<=	Kleiner oder gleichwertig
*	Multiplikation	!=	Ungleichheit
/	Division	=	Gleichheit
^	Potenz		

Diese Operationen können verbinden Operanden. So sind komplexere Bedingungen möglich, die eine breite Abdeckung der Anforderungen sicherstellen.

Verkettungen

Mit dem Grundgerüst der im vorherigen Absatz besprochenen Operationen lassen sich Bedingungen erstellen. Eine einzige Bedingung pro Breakpoint ist aber nicht immer ausreichend und eine unnötige Limitation. Aus diesem Grund inkludierten wir zwei weitere Operationen, die den Nutzenden ermöglichen, mehr als eine Bedingung mit einer einzigen Zeichenkette zu erstellen.

Mit den zwei zusätzlichen Literalen «AND» und «OR» können zwei oder mehr Bedingungen miteinander verkettet werden. Die Zeichenkette «RAX = 488B042500000000 AND RBX = 0» wird analog zur folgenden Grafik aufgelöst:

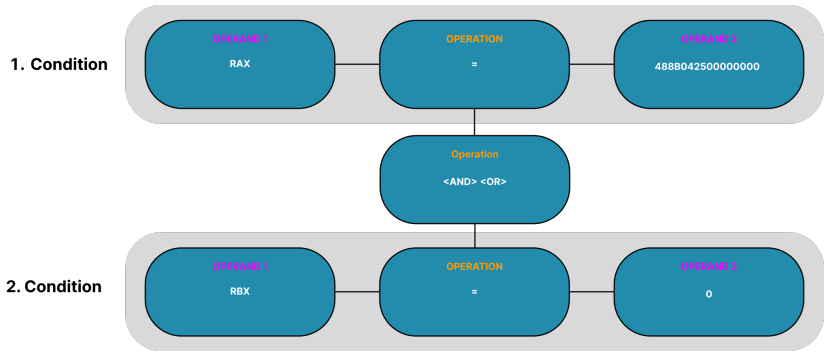

Abbildung 15 - Interpretation des Strings

Der Term wird vom Parser in der von uns festgelegten Reihenfolge verarbeitet.

Weiterentwicklungspotential: In unserer Implementation des Expression Parsers könnten noch weitere Operationen, wie zum Beispiel «XOR» erstellt werden. Möglicherweise besteht auch das Bedürfnis, weitere Typen von Operanden zur Verfügung zu haben. Ein Beispiel hierfür könnten spezifische Speicher-Adressen sein, die in Bedingungen verglichen werden könnten.

Typenlimitierung

Beim Arbeiten mit Registerwerten begegneten wir einem entscheidenden Problem. Um das Beispiel von vorher erneut zu betrachten:

```
RAX = 488B042500000000
```

Der Wert des Registers ist ausserordentlich lang und ausserdem ein Hex-Wert. Weder JavaScript noch Typescript sind in der Lage, mit hexadezimalen Werten zu rechnen. Der Wert muss zuerst in den Datentyp «Integer» (**int**) konvertiert werden:

```
parseInt(hexNumber, 16)
```

Aus «488B042500000000» wird so der Dezimalwert:

```
RAX = 5227276349453894000
```

Wenn wir die Konvertierung jedoch von Hand vornehmen, führt das zu folgendem Resultat:

$$(488B042500000000)_{16} =$$
$$(4 \times 16^{15}) + (8 \times 16^{14}) + (8 \times 16^{13}) + (11 \times 16^{12}) +$$
$$(0 \times 16^{11}) + (4 \times 16^{10}) + (2 \times 16^{9}) + (5 \times 16^{8}) +$$
$$(0 \times 16^{7}) + (0 \times 16^{6}) + (0 \times 16^{5}) + (0 \times 16^{4}) +$$
$$(0 \times 16^{3}) + (0 \times 16^{2}) + (0 \times 16^{1}) + (0 \times 16^{0}) =$$
$$(5227276349453893632)_{10}$$

Die Dezimalwerte sind zwar ähnlich, aber nicht identisch.

```
5227276349453894000 != 5227276349453893632
```

Diese Abweichung ist deshalb zu erklären, weil der **Integer**-Datentyp in JavaScript [21] maximal den Dezimalwert 9'007'199'254'740'992 (= 2^{53}-1) annehmen kann. Insgesamt sind das 16 Stellen, was auch erklärt, warum die letzten drei Stellen unserer Beispielzahl 5’227’276’349’453’894’000 mit null aufgefüllt werden und die viertletzte Stelle schlicht aufgerundet wird.

Wenn man diese Limitierung umgehen möchte, ist das Verwenden des Datentyps **BigInt** [22] notwendig, da dieser auch mit noch grösseren Zahlen umgehen kann.

```
const hexStringToBigInt = (input) => BigInt(`0x${input}`);
console.log(hexStringToBigInt(number).toString())
= “5227276349453893632”
```

Unglücklicherweise verwendet die von uns verwendete Expression Parser Library den regulären JavaScript-Datentyp **Number**, darunter fällt auch der **Integer**-Datentyp. Wir mussten deshalb spezielle Typen anlegen und die bestehende Typendeklaration erweitern, um **BigInt** statt **Number** verwenden zu können. An der Logik der Library selbst mussten wir nichts anpassen, da der Programmcode in JavaScript konvertiert wird und Standard-Operationen wie +, -, = oder / auf **BigInt** ebenfalls existieren.

Validation

Die Bedingungen können nun interpretiert und ausgewertet werden. Die Konvertierung von «RAX» in einen Zahlenwert kann allerdings erst beim Auslösen des Breakpoints erfolgen – der Inhalt des Registers könnte sich ja bis dahin noch ändern. Trotzdem müssen Nutzende bereits beim Setzen des Breakpoints darüber informiert werden, ob ihre Bedingung korrekt formuliert ist. Wer zum Beispiel versucht, auf den Register «RKX» zuzugreifen, wird keinen Erfolg haben, da dieser nicht existiert. Es wäre aber wenig sinnvoll und frustrierend, den Breakpoint einfach nie auszulösen, weil ein Operand nicht gefunden werden kann. Wir möchten Nutzenden stattdessen bereits beim Schreiben der Bedingung eine Fehlermeldung anzeigen.

Damit wir über die Richtigkeit von Eingaben informieren können, haben wir eine Validations-Erweiterung erstellt.

Die Validation funktioniert sehr analog zum eigentlichen Parser. Es wird für jedes Element überprüft, ob es sich um eine Operation oder einen Operanden (Zahl, Register oder Flag) handelt. Falls ein Register oder Flag genannt wird, wird ein Platzhalterwert an dieser Stelle eingesetzt:

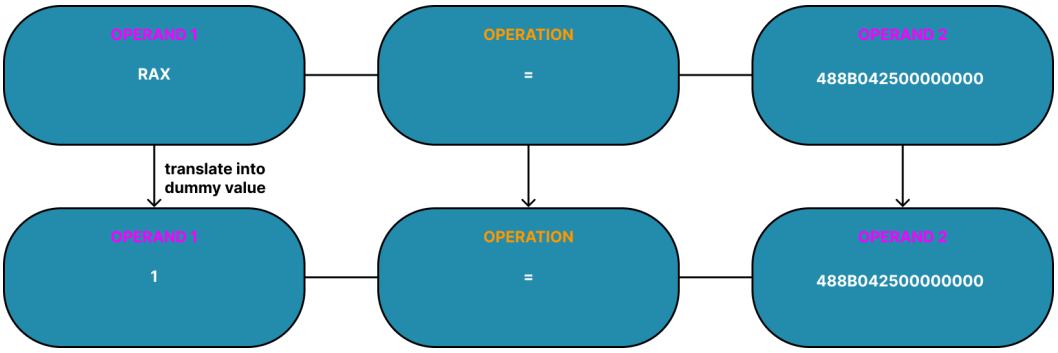

Abbildung 16 - Translation Operand 1 in einen Platzhalterwert

Mit diesem Platzhalterwert kann die Bedingung weiterverarbeitet werden und die Validation kann überprüfen, ob als Ergebnis **true** oder **false** herauskommt. Das Ergebnis stimmt natürlich nicht und wird sofort wieder verworfen, aber es geht schliesslich nur um das Prüfen der Semantik.

Als nächstes wollten wir das Feedback gegenüber Nutzenden im Falle einer fehlgeschlagenen Validierung durch das Anzeigen von Fehlermeldungen verbessern.

Fehlermeldungen

Das Erstellen der Bedingungen wird mittels eines Formulars ermöglicht. Zum Einsatz kommt eine dynamische Fehlermeldung, die direkt dargestellt werden kann. Erst wenn keine Fehler mehr vorhanden sind, kann die Bedingung abgespeichert werden. Die verschiedenen Meldungen werden an diversen Stellen im Validationsteil des Parsers erstellt und von der Benutzeroberfläche dargestellt.

Zum Zeitpunkt des Schreibens dieser Arbeit existieren folgende Fehlermeldungen:

Fehlermeldung	**Grund**
'Please enter your condition.'	Die Eingabe ist leer.
'Missing at least one other operand or value'	Es fehlt mindestens ein Operand oder eine Konstante. Beispiel: ‘RAX + + 1 = RBX’ Erklärung: Es fehlt ein Operand (Register/Flag oder Hex-Wert) zwischen den beiden ‘+’ - Zeichen.
<X> is not a valid operand. Use registers, flags or constants.'	Register oder Flag nicht gefunden. Beispiel: ‘RKX = RAX’ Erklärung: RKX ist kein valides Register
<X, Y, …> are not valid operands. Use registers, flags or constants.	Analog zur vorhergehenden Meldung, jedoch sind mehrere Strings betroffen. Beispiel: ‘RKX + RFX = RAX’ Erklärung: RKX und RFX sind beides keine gültigen Register.
'Your Condition does not evaluate to TRUE or FALSE'	Wenn keine sonstigen Fehlermeldungen mehr aktiv sind, wird die Bedingung validiert. Wenn danach der Rückgabewert nicht **true** oder **false** entspricht, erscheint die linke Meldung.

‘Your arrangement of operations and operands is formally not correct’	Das Verhältnis von Operationen und Operanden ist nicht gültig. Beispiel: ‘AL AL =’ Erklärung: Es fehlt eine Operation zwischen AL und AL und ebenso ein Operand, mit dem verglichen werden soll.

Diese Fehlermeldungen sollten ausreichend sein, um den Nutzenden zu vermitteln, inwiefern sie ihre Eingabe anpassen müssen, um eine korrekte Bedingung zu erhalten. Falls in Zukunft weitere Fehlermeldungen gewünscht sind, können diese leicht hinzugefügt werden.

4.3. Implementation Frontend

Anfangs wollten wir für die Eingabe der Bedingung eine Benutzeroberfläche konstruieren, die es Nutzenden per Drag & Drop oder per Dropdown-Auswahl ermöglicht, Bedingungen sozusagen «zusammenzubauen». Der Vorteil hiervon ist, dass der User sofort weiß, welche Elemente er verwendet darf und wie eine Bedingung aufgebaut werden muss. Zu diesem Zwecke evaluierten wir zwei Libraries: zuerst sahen wir uns *query-builder-vue* [23] an.

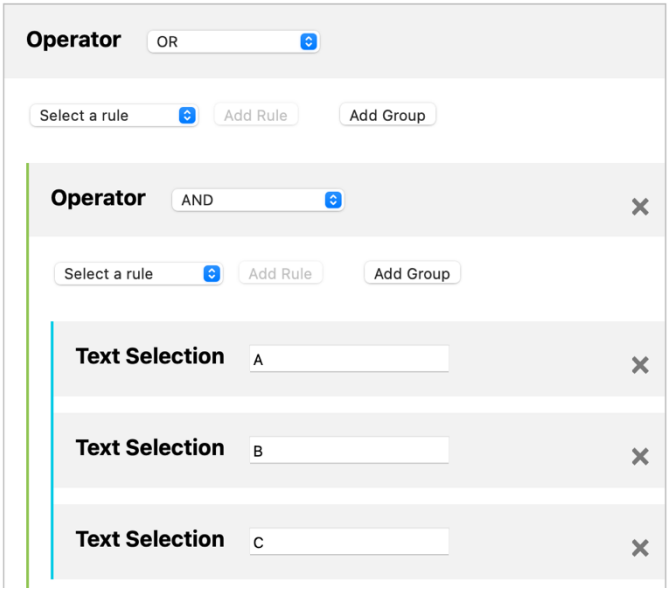

Abbildung 17 - Die Library query-builder-vue

Der entscheidende Vorteil dieser Library war, dass sie Vue unterstützt und daher vermutlich einfach in unsere bestehende Architektur eingebunden werden konnte. Ausserdem schien sie die von uns gewünschte Funktionalität zu erfüllen. Wir bemerkten allerdings bei der Einbindung schnell, dass Query-Builder in Vue 2 geschrieben war. Wir hatten unser Projekt aber inzwischen auf Vue 3 migriert (siehe Kapitel **Migration**). Da Vue 3 nicht rückwärtskompatibel ist, machte die Modernisierung der Applikation die einfache Verwendung dieser Library also ironischerweise unmöglich.

Daher sahen wir uns nach einer Alternative um und stiessen dabei auf *jQuery QueryBuilder* [24]:

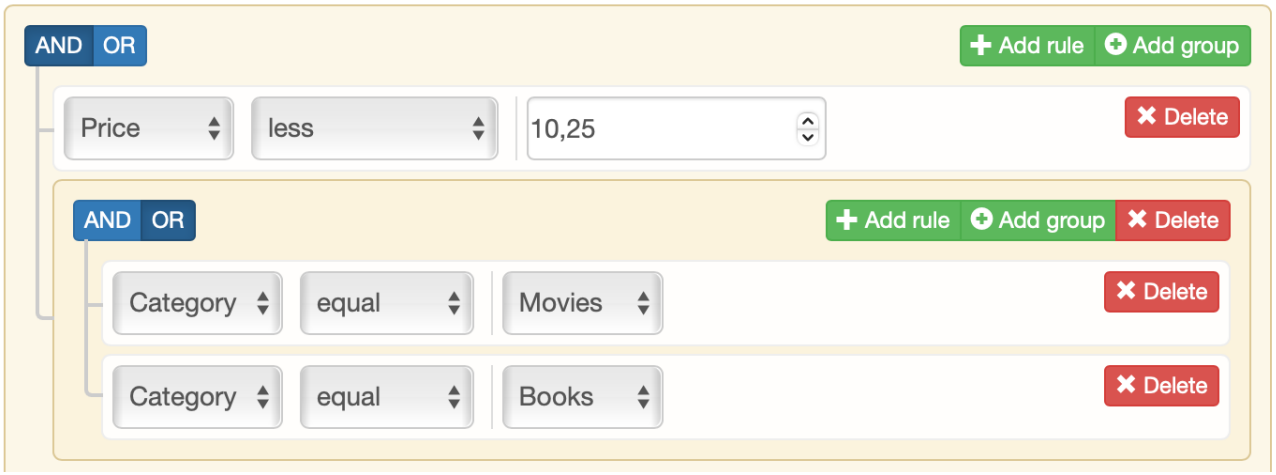

Abbildung 18 - Die Library jQuery QueryBuilder

Beim Versuch, diese einzubinden, stellten wir fest, dass dies einigen Overhead in der Form von Abhängigkeiten mit sich brachte. Insbesondere die Library *jQuery* ist über einen Megabyte gross. Da unser Simulator zu Beginn der Arbeit insgesamt nur 7MB gross war, würde *jQuery* den Speicher-Fussabdruck also deutlich erhöhen. Da wir es nur für dieses eine Feature benützen würden, kam uns dies unverhältnismässig vor.

Wir besprachen die Situation mit Prof. Richter und er empfahl uns, User einfach einen String zur Definition der Bedingung eingeben zu lassen. Dadurch erhält man maximale Flexibilität und die Bedingung kann völlig frei spezifiziert werden, ohne sich an ein neues GUI (das zudem nicht im Stil des restlichen Simulators entworfen ist) gewöhnen zu müssen.

Als nächstes stellte sich die Frage, wie Nutzende einen solchen bedingten Breakpoint setzen könnten. Wir wollten das Feature möglichst organisch über die bestehende Benutzeroberfläche zugänglich machen. Ausserdem sollte es thematisch mit dem bereits implementierten Prozess des Setzens

eines normalen Breakpoints verknüpft werden. Daher fügten wir einen Button oberhalb der Assembly-Code-Anzeige ein. Als Icon wählten wir eine Checkliste aus den Google Material Icons [16], da sie an die zeilenweise Überprüfung von Bedingungen erinnert. Da ein bedingter Breakpoint allerdings auf einer spezifischen Zeile gesetzt wird, mussten wir auch das Klicken auf die Zeilennummer beibehalten. Daher agiert der Button als eine Art Schalter und informiert die Applikation, dass der nun folgende Breakpoint eine Bedingung haben soll. Ein Tooltip informiert beim Mouseover über den Button über diesen Umstand. Zur besseren Verständlichkeit färbt er sich

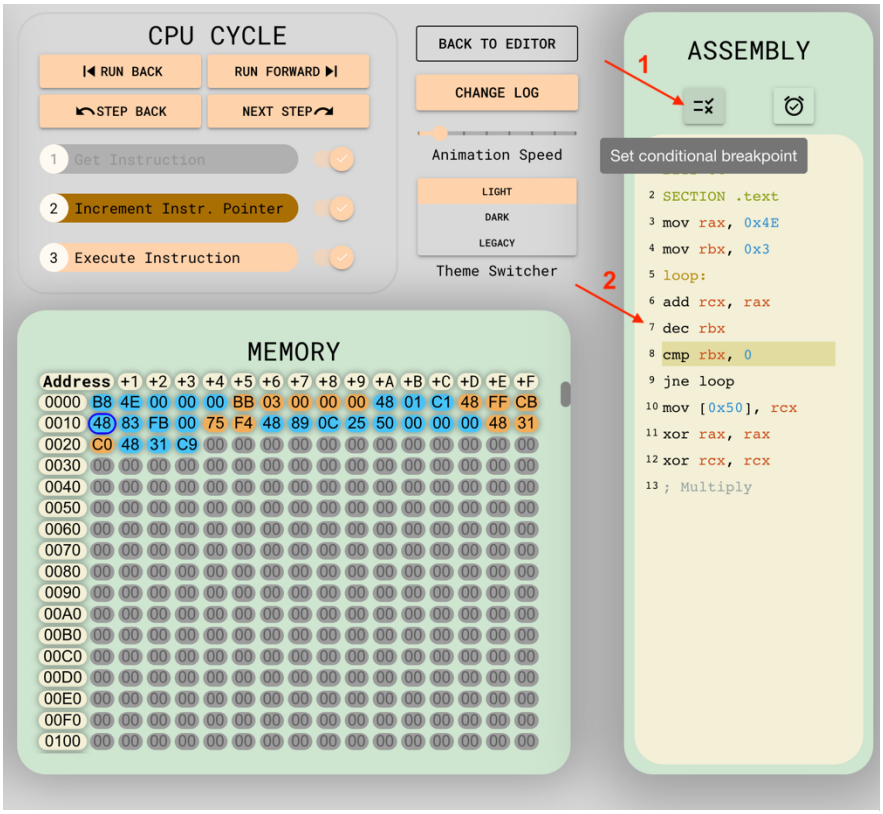

Abbildung 19 - Setzen eines bedingten Breakpoints

nach dem Klick ausserdem blau ein, um die Aktivierung des Modus «bedingter Breakpoint» zu signalisieren. Gleichzeitig erscheint ein Hilfstext am unteren Rand des Bildschirms. Dieser weist Nutzende an, nun auf eine Zeilennummer zu klicken, um damit einen bedingten Breakpoint zu erstellen. Tun sie dies, öffnet sich ein Dialogfenster, das die Eingabe eines Strings erlaubt. Später wurde dieses Fenster in seiner Funktionalität deutlich erweitert (siehe Abschnitt **5.4**). Der gesamte Prozess ist in **Abbildung 19** ersichtlich.

Es stellte sich ausserdem die Frage nach dem Verhalten bei einem normalen Klick auf eine Zeilennummer, auf der bereits ein Breakpoint existiert. Bisher entfernte ein solcher Klick bestehende Breakpoints oder fügte andernfalls neue hinzu (Toggle). Wir überlegten uns, dass das Entfernen von bedingten Breakpoints ebenfalls durch einen einfachen Klick möglich sein sollte. Falls der Nutzende aber den in **Abbildung 19** dargestellten Ablauf auf einer Zeile mit gesetztem Breakpoint ausführt, sollte ein neuer bedingter Breakpoint auf der Zeile erstellt werden und dabei etwaige bereits existierende ersetzen (egal ob bedingt oder unbedingt). Damit stellen wir sicher, dass Nutzereingaben im Bedingungsfeld nicht umsonst erfolgen. Somit ergibt sich folgendes Verhalten:

Button «bedingter Breakpoint» aktiviert	**Klick auf…**	**Effekt**
Nein	Leere Zeile	Unbedingter Breakpoint wird erstellt
Nein	Zeile mit unbedingtem Breakpoint	Bestehender Breakpoint wird gelöscht
Ja	Leere Zeile / Zeile mit Breakpoint beliebigen Typs	Bedingter Breakpoint wird erstellt. Gegebenenfalls bereits bestehender Breakpoint wird gelöscht

Nun fehlte nur noch eine visuelle Differenzierung der bedingten Breakpoints von den normalen. Hierfür erachteten wir ein Fragezeichen neben dem Breakpoint-Symbol als angebracht, da es die optionale Natur der Sache passend widerspiegelt. Zur Erklärung informiert ein Tooltip beim Mouseover über die Bedeutung des Fragezeichens.

Für das Hinzufügen des Fragezeichens musste die in den Abschnitten **1.3** und **3.3** erwähnte HTML-Tabelle um eine zusätzliche Spalte erweitert werden. Diese Spalte ist normalerweise leer und damit nicht sichtbar, wird aber beim Vorhandensein mindestens eines Breakpoints angezeigt. Als wir dies Prof. Richter zeigten, schlug er zusätzlich die Verwandlung des Fragezeichens in ein Ausrufezeichen vor, falls der Breakpoint ausgelöst wird. Wir implementierten dies und fügten ausserdem noch einem neuen Infotext am unteren Ende des Bildschirms hinzu, der die Nutzenden darüber informiert, dass ein bedingter Breakpoint ausgelöst wurde und welche

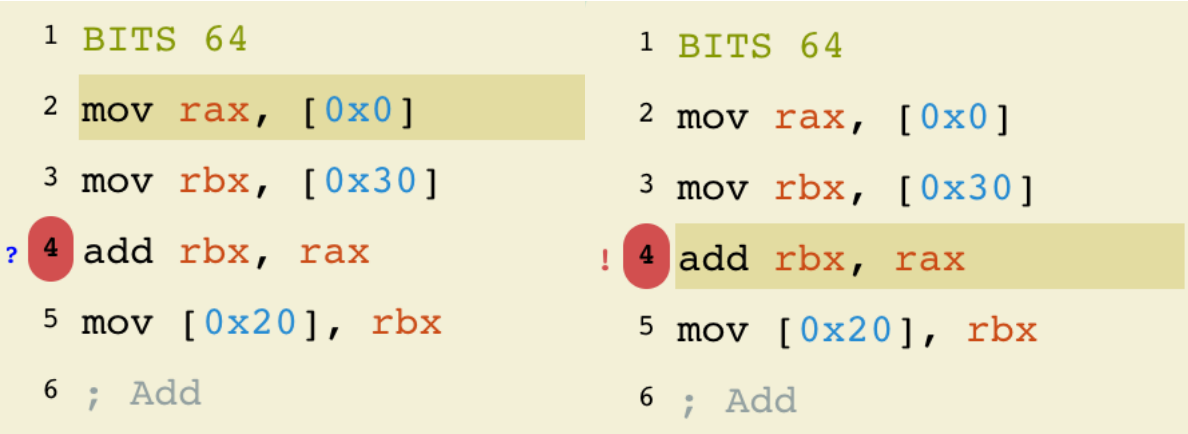

Abbildung 20 - Ein gesetzter (links) und ein ausgelöster Breakpoint (rechts)

Bedingung ihn ausgelöst hat. Auch der Tooltip über dem Ausrufezeichen sagt aus, dass es sich um einen ausgelösten Breakpoint handelt. Nach 5 Sekunden wird sowohl das Ausrufezeichen entfernt als auch der Tooltip auf den vorherigen Text zurückgesetzt.

5. Watchpoints

5.1. Motivation

Ein Watchpoint ist einem bedingten Breakpoint sehr ähnlich. Es wird eine Bedingung angegeben, unter der der Programmfluss angehalten werden soll. Unterschieden wird hierbei jedoch beim Zeitpunkt, zu dem die Bedingung überprüft wird. Bei bedingten Breakpoints wird die Bedingung ausschliesslich beim Erreichen der Breakpoint-Zeile ausgewertet, auf der der Breakpoint gesetzt wurde. Bei Watchpoints hingegen wird die Bedingung nach jedem Schritt ausgewertet:

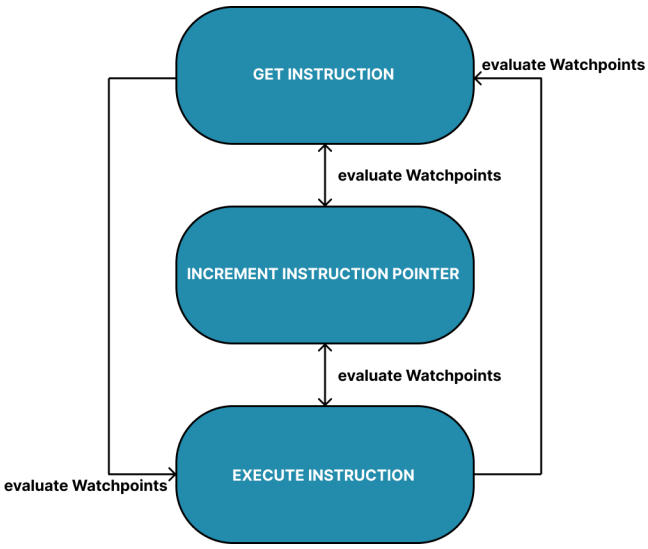

Abbildung 21 - Auswertung von Watchpoints

Ein Szenario, in dem Watchpoints nützlich sein können, ist das Beobachten eines spezifischen Speicherbereichs bei der Ausführung eines komplexen Programms. Es ist gut möglich, dass man als Programmierer bemerkt, dass dieser Speicher an einem bestimmten Punkt den falschen Wert enthält. Aufgrund der Komplexität des Programms ist aber unklar, wann dieser falsche Wert dort landet und welche Instruktion ihn dort hineinschreibt. Durch Setzen eines Watchpoints kann jegliche Veränderung dieses Speicherbereichs überwacht werden und das Programm an geeigneter Stelle für eine Analyse angehalten werden.

5.2. Implementierung Backend

Wir können unseren Condition-Parser erneut für die Watchpoints verwenden. Das schafft bessere Wartbarkeit und Kohärenz für die Nutzenden.

Nachdem ein Watchpoint über die grafische Benutzeroberfläche gesetzt wurde, fügen wir ihn in ein Array ein (analog zum Vorgehen bei Breakpoints). Ein Watchpoint wird von der Logik gleich wie ein Breakpoint verwendet:

```
interface Watchpoint {
  condition: Condition;
}
```

Die Watchpoints müssen nach jedem Schritt überprüft werden. Wir erreichten dies mit einem Wrapper, den wir um die Funktionen *nextStep* und *stepBack* angelegt haben. Er führt den jeweiligen Schritt aus und fragt beim «debuggerController» an, ob Watchpoints gesetzt sind. Falls ja, wird jede Bedingung überprüft und falls die Bedingung wahr ist, wird eine Nachricht an das Frontend gesendet.

5.3. Implementierung Frontend

Die Watchpoints sind jetzt funktionsfähig und einsatzbereit. Im Gegensatz zu den Breakpoints können wir diese den Nutzern aber nicht grafisch pro Zeilennummer darstellen, da Watchpoints nicht an individuelle Zeilen gekoppelt sind. Es war deshalb nötig, eine Übersicht zu erstellen, in der alle Watchpoints aufzufinden sind – damit haben Nutzende einen Überblick und können auf Wunsch auch Einträge entfernen.

Um die Nutzeroberfläche möglichst einfach zu halten, fügten wir lediglich einen zusätzlichen Button neben dem bereits bestehenden Button für bedingte Breakpoints ein.

Beim Klick darauf öffnet sich in der Mitte des Browsers ein Dialogfenster, das die obengenannten Funktionen erfüllt:

Abbildung 22 - Interaktiver Watchpoint Button

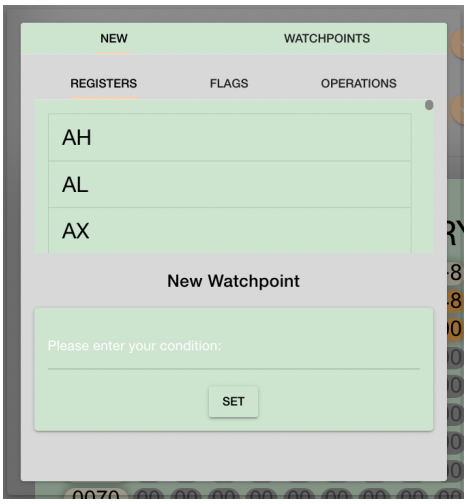

Abbildung 23 - Watchpoint Fenster, Tab "NEW"

Das Fenster besteht aus den zwei Tabs «NEW» und «WATCHPOINTS».

Tab NEW

In «NEW» kann ein neuer Watchpoint angelegt werden. Der Tab ist weiter in zwei Bereiche gegliedert. Das untere Drittel befasst sich mit dem Erfassen eines neuen Watchpoints.

Textfeld

Beim Interagieren mit dem Textfeld wird direkt zurückgegeben, ob die Eingabe fehlerhaft ist. Ausserdem werden hier die in Abschnitt **4.2** erwähnten Fehlermeldungen angezeigt:

Abbildung 24 - Beispiel einer Fehlermeldung

Werden keine Fehler vom Parser gefunden, leuchtet das Textfeld in den Farben des aktuell aktiven Farbschemas auf und die rote Fehlermeldung verschwindet:

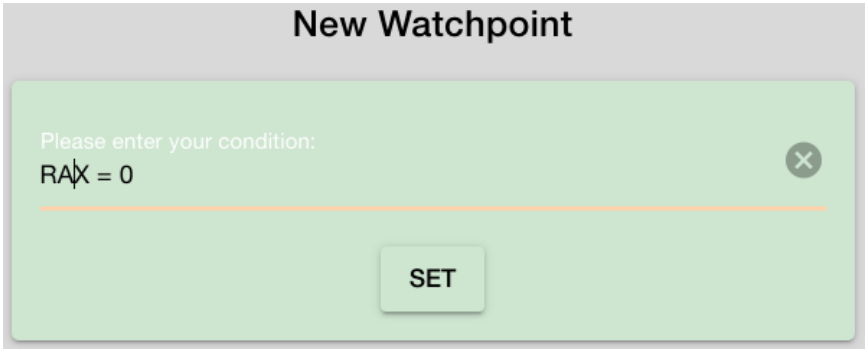

Abbildung 25 - Beispiel einer korrekten Bedingung

Beim Klicken von «SET» wird der Watchpoint gesetzt und das Dialogfenster schliesst sich.

Operanden & Operationen

Der obere Teil des Tabs setzt sich aus drei weiteren Tabs zusammen. In diesen Tabs enthalten sind Register, Flags und Operationen. Es sind individuelle Auflistungen aller möglichen Zeichenketten und Operationen, die vom Parser verstanden und verarbeitet werden können.

Die Tabelle kann ebenfalls angeklickt werden. Das Register, die Flag oder die Operation fügen sich danach in die unten angezeigte Textbox ein:

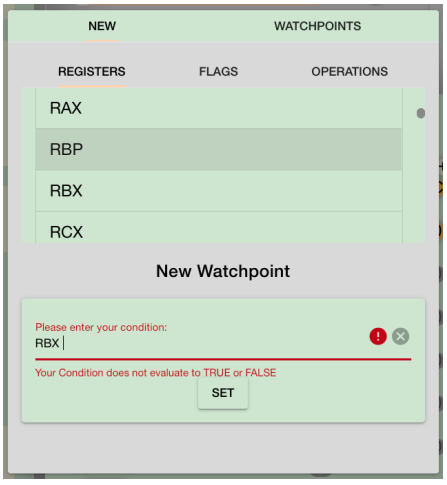

Abbildung 27 - Einfügen des Tabelleneintrags in die Textbox

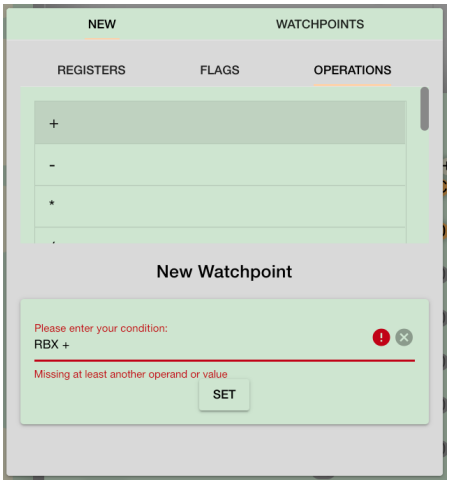

Abbildung 26 - Anhängen einer Operation in die Textbox

Tab WATCHPOINTS

Der Tab «WATCHPOINTS» listet alle gesetzten Watchpoints in einer Tabelle auf:

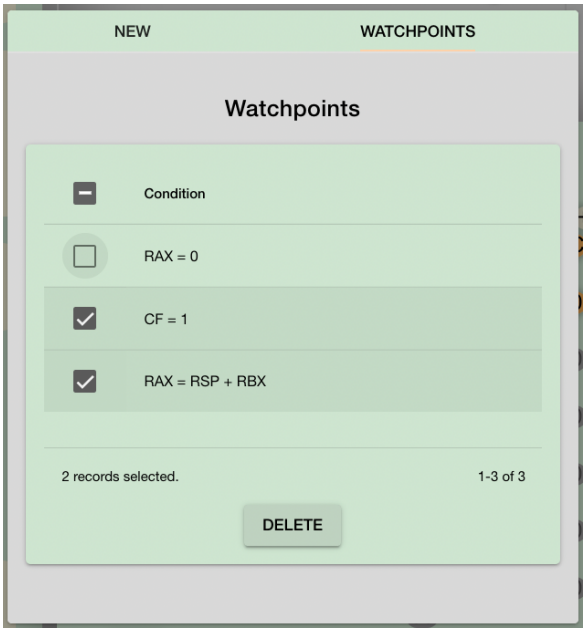

Abbildung 28 - Selektierung mehrerer Watchpoints

Die links angebrachten Auswahlfelder können angewählt werden, um den Watchpoint zu selektieren. Mehrfachauswahl ist möglich.

Über den «DELETE» Button werden alle ausgewählten Watchpoints entfernt. Leer sieht die Tabelle wie folgt aus:

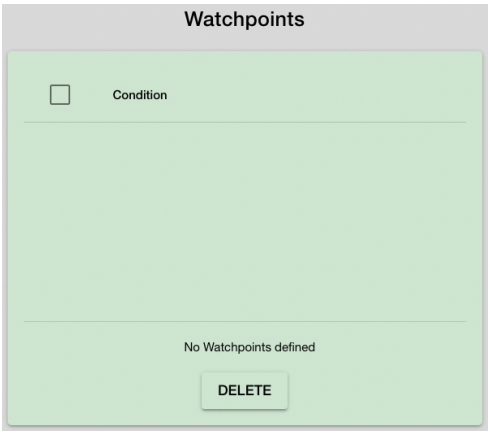

Abbildung 29 - Leere Tabelle mit Zustandsmeldung

Zukünftige Verbesserungen: Falls die Applikation weiterentwickelt werden soll, wäre es sicher sinnvoll, die Watchpoints hier direkt in der Tabelle editierbar zu machen. Sollte sich ein Nutzender nämlich beim Eintippen der Bedingung vertippen, muss in der aktuellen Version der Watchpoint entfernt und neu angelegt werden.

5.4. Rückportierung UI für bedingte Breakpoints

Die Listen-Übersicht und das überarbeitete Userinterface waren nun erstellt und wir waren zufrieden mit dem Ergebnis. Daher wollten wir diese neue UI-Komponente als nächstes auch für unseren Prototypen zum Erstellen von bedingten Breakpoints (der zu diesem Punkt nur aus einem Dialogfenster mit einem Textfeld bestand) übernehmen. Alle Breakpoints sind zwar bereits im Code Viewer sichtbar: kontextuell gesetzte Farben und wechselnde Icons zeigen auf, wo Breakpoints gesetzt und ausgelöst werden. Der Nachteil an diesem Ansatz ist allerdings, dass die Bedingungen gesetzter Breakpoints nicht mehr angezeigt werden können. Das neue UI würde dieses Problem beheben und ausserdem das Löschen mehrerer bedingter Breakpoints auf einmal erlauben.

Wir wollten deshalb ebenfalls eine Tabelle analog zu der mit den Watchpoints implementieren. Um Duplikation von Code zu vermeiden, haben wir aus dem oben beschriebenen Feature eine dynamische Komponente erstellt.

Konfiguration für Watchpoints:

```
<conditional-configurator
  :lookup-function="reverseLookup"
  :title="'Watchpoints'"
  :conditionals="watchpoints"
  :set-conditional-function="setConditionalWatchpoint"
  :delete-conditional-function="deleteWatchpoints"
/>
```

Konfiguration für Breakpoints:

```
<conditional-configurator
  :lookup-function="reverseLookup"
  :title="'Breakpoints'"
  :conditionals="breakpoints"
  :set-conditional-function="setConditionalBreakpoint"
  :delete-conditional-function="deleteBreakpoints"
/>
```

Der «conditional-configurator» baut die grafischen UI-Elemente entsprechend der übergebenen Funktionen und Datensätze auf.

Das Endprodukt ist eine wiederverwendbare Komponente ohne duplizierten Code:

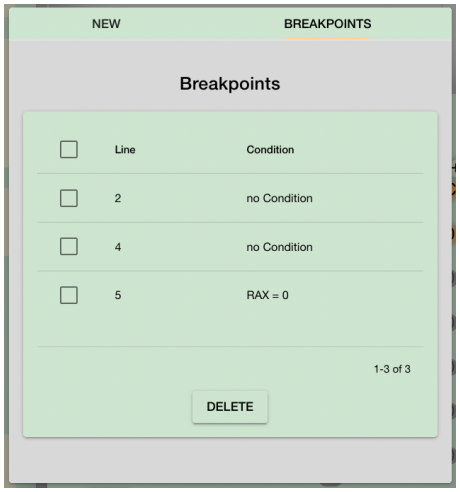

Abbildung 30 - Auflistung aller Breakpoints

6. Sprung zu spezifischer Zeile

6.1. Motivation

Wie bereits erwähnt, wollten wir die Interaktionen mit dem eigenen Code noch intuitiver gestalten. Wenn man eine Instruktion besonders interessant findet, ist es naheliegend, dass man sie anklickt. Wir dachten uns, dass es sinnvoll wäre, wenn das Programm daraufhin zur Ausführung der betreffenden Instruktion springt. Somit könnte man eine besonders interessante Instruktion direkt „in Aktion“ sehen, ohne vorher dort einen Breakpoint setzen zu müssen.

6.2. Implementation Backend

Dank der in Kapitel **3** implementierten Funktionen war dieses Feature sehr leicht einzufügen. Das Frontend liefert die Zeilennummer, auf die gesprungen werden soll. Dann wird die Funktion *runUntil* (die normalerweise verwendet wird, um zum nächsten/vorherigen Breakpoint zu laufen) mit der Zeilennummer als Argument aufgerufen und die Ausführung springt zum entsprechenden Punkt.

6.3. Implementation Frontend

Auch in der Benutzeroberfläche war das Feature einfach zu implementieren. Die bereits mehrfach erwähnte HTML-Tabelle wurde erweitert, sodass ein Klick auf eine Zeile Code eine Funktion aufrief. Diese nimmt die Zeilennummer, konvertiert sie mittels der *translationTable* in eine Backend-Zeilennummer und erstellt dann einen Event, der die eingangs erwähnte Funktion im Backend aufruft. Der Sprung an die neue Stelle im Code erfolgt dabei genau wie bei «Run Forward» ohne Animationen. Watchpoints werden dabei überprüft, Breakpoints aber nicht. Der Hintergedanke war folgender: Breakpoints sind im Code Viewer selbst sichtbar und der Nutzende hat ja seine Aufmerksamkeit in diesem Moment auf dem Code Viewer. Daher dürfte ihm das Überspringen von Breakpoints bewusst sein. Watchpoints hingegen sind nur in der Tabelle des Dialogfensters sichtbar. Deshalb lösen wir sie beim Sprung vorwärts oder rückwärts weiterhin aus, um gegebenenfalls auf ihre erfüllte Bedingung hinzuweisen.

Migration

1. Motivation

Die Arbeit am CPU-Simulator begann vor zwei Jahren. Die verwendeten Technologien, Abhängigkeiten und Bibliotheken stammen aus dieser Zeit. Uns fiel auf, dass einige der zusätzlich verwendeten NPM-Packages, wie beispielsweise die Vue Class Component Library [25] und einige der Webpack Libraries nicht mehr weiter entwickelt werden und als deprecated (= veraltet) gelten. Veraltete Software kann Sicherheitslücken beinhalten und schränkt Programmierer bei der Verwendung von neueren Technologien ein.

Wir wollten unser Projekt in einem möglichst aktuellen und wartungsfreundlichen Zustand übergeben – schliesslich ist nicht ausgeschlossen, dass weitere Generationen von Studenten daran arbeiten werden. Aus diesem Grund beschlossen wir, diverse Bibliotheken (unter anderem Vue und Quasar) auf die neueste Version zu aktualisieren.

```
➜  x86 git:(e8f4cb9) ✗ npm install
npm does not support Node.js v15.11.0
You should probably upgrade to a newer version of node as we
can't make any promises that npm will work with this version.
You can find the latest version at https://nodejs.org/
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: '@achrinza/node-ipc@9.2.2',
npm WARN EBADENGINE   required: { node: '8 || 10 || 12 || 14 || 16 || 17' },
npm WARN EBADENGINE   current: { node: 'v15.11.0', npm: '8.5.0' }
npm WARN EBADENGINE }
npm WARN deprecated source-map-url@0.4.1: See https://github.com/lydell/source-map-url#deprecated
npm WARN deprecated request-promise-native@1.0.9: request-promise-native has been deprecated because it extends the now deprecated request package,
npm WARN deprecated @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/topo@3.1.6: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated eslint-loader@2.2.1: This loader has been deprecated. Please use eslint-webpack-plugin
npm WARN deprecated mkdirp@0.5.4: Legacy versions of mkdirp are no longer supported. Please update to mkdirp 1.x. (Note that the API surface has cha
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated source-map-resolve@0.5.3: See https://github.com/lydell/source-map-resolve#deprecated
npm WARN deprecated debug@3.2.6: Debug versions >=3.2.0 <3.2.7 || >=4 <4.3.1 have a low-severity ReDos regression when used in a Node.js environment
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm WARN deprecated chokidar@2.1.8: Chokidar 2 does not receive security updates since 2019. Upgrade to chokidar 3 with 15x fewer dependencies
npm WARN deprecated chokidar@2.1.8: Chokidar 2 does not receive security updates since 2019. Upgrade to chokidar 3 with 15x fewer dependencies
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm WARN deprecated chokidar@2.1.8: Chokidar 2 does not receive security updates since 2019. Upgrade to chokidar 3 with 15x fewer dependencies
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchParams API instead.
npm WARN deprecated html-webpack-plugin@3.2.0: 3.x is no longer supported
npm WARN deprecated @hapi/address@2.1.4: Moved to 'npm install @sideway/address'
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain circumstances, which is kno
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated @hapi/hoek@8.5.1: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/joi@15.1.1: Switch to 'npm install joi'
npm WARN deprecated svgo@1.3.2: This SVGO version is no longer supported. Upgrade to v2.x.x.
npm WARN deprecated tar@2.2.2: This version of tar is no longer supported, and will not receive security updates. Please upgrade asap.
npm WARN deprecated core-js@2.6.12: core-js@<3.4 is no longer maintained and not recommended for usage due to the number of issues. Because of the V
he actual version of core-js.

added 1769 packages, and audited 1770 packages in 2m

116 packages are looking for funding
  run `npm fund` for details

34 vulnerabilities (3 moderate, 23 high, 8 critical)

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Abbildung 31 - Warnungen des Package Managers NPM nach Installation aller Dependencies

Es folgt eine Auflistung der wichtigsten aktualisierten Packages:

	Version		
Paket	**Vorher**	**Neu**	**Beschreibung**
Quasar	1.0.0	2.6.5	Grafische Userkomponenten
Vue	2.6.11	3.2.13	Web-Framework
Vue-Router	3.2.0	4.0.3	Routing innerhalb des Simulators
Vue-Class-Component	7.2.3	Entfernt (veraltet)	Klassenähnliche Logik für Vue-Komponenten
Vue-Prism-Editor	1.2.2	2.0.0-alpha.2	Code-Editor, um den Assembly Code zu verändern
Typescript	3.9.3	4.5.5	Unsere Programmiersprache
Chai	4.1.2	4.2.0	Testutensilien
Webpack	4	5	Packt alle Packages in eine auslieferbare Webapplikation

2. Notwendige Veränderungen

Ursprünglich planten wir, ein minimales Upgrade vorzunehmen. Wir wollten die Sicherheitslücken schliessen, ohne dabei neue «Major Releases» zu installieren, da diese nicht immer rückwärtskompatibel sind [26].

Nach diversen Anläufen gelang uns dies nicht. Der Hauptgrund dafür ist Webpack [27]. Viele Bibliotheken referenzieren und benötigen Webpack. Zu Beginn des Projektes wurde Webpack 4 verwendet. Kurz danach wurde allerdings Webpack 5 veröffentlicht, und viele der anderen von uns verwendeten Libraries wechselten auf diese Version für ihre Releases. Webpack 5 ist nicht rückwärtskompatibel zu Version 4.

Wir haben uns deshalb entschlossen, die Migration auf Major Versions doch bei allen Packages vorzunehmen.

Die grösste und bedeutendste Änderung hierbei war das Entfernen der Vue-Class-Component Library. Gemäss zweier Github Issues [28][29] wird nicht empfohlen, die Library weiterhin zu benutzen. Gemäss dem Package-Manager NPM wurde die Library seit mehr als zwei Jahren nicht mehr aktualisiert [25].

Version History

Version	Downloads (Last 7 Days)	Published
8.0.0-rc.1	34,397	2 years ago
8.0.0-beta.4	61	2 years ago
8.0.0-beta.3	106	2 years ago
7.2.6	367,285	2 years ago
8.0.0-beta.2	6	2 years ago
8.0.0-beta.1	3	2 years ago

Abbildung 32 - Übersicht über alle Versionen der Vue-Class-Component Library

Mit dem Entfernen dieser Library entstand die Notwendigkeit, die betroffenen Vue-Komponenten umzuschreiben.

2.1. Refactoring der Vue Komponenten

Bis anhin wurde ein klassenbasierter Ansatz mit Vue-Class-Component verfolgt, analog zu diesem Beispiel:

```
@Component({
  components: {
    MemoryLine,
  },
})
export default class MemoryAreaData extends Vue {
  @Prop()
  memoryData!: MemoryData;

  @Prop()
  byteInformation!: ByteInformation;
}
```

Neu mussten wir uns für eine der beiden API-Ansätze entscheiden, die Vue 3 anbietet. Diese APIs sind die «Composition API» [30] und «Options API» [31]. Das Vue Developer Team empfiehlt stark, erstere zu wählen, sofern das Projekt in Typescript geschrieben wird [32]:

“While Vue does support TypeScript usage with Options API, it is recommended to use Vue with TypeScript via Composition API as it offers simpler, more efficient and more robust type inference.” [33]

Die gleiche Komponente mit Vue 3 Composition API sieht wie folgt aus:

```
export default defineComponent({
  name: 'MemoryAreaData',
  components: {
    MemoryLine,
  },
  props: {
    memoryData: { type: Object as PropType<MemoryData>, required: true },
    byteInformation: { type: Object as PropType<ByteInformation>, required:
false },
  },
});
```

In der Composition API wird die Komponente in den «defineComponent» Wrapper gewickelt. Darin definiert sind:

Name	**Name der Komponente**
components	Abhängige Komponente innerhalb dieser Komponente
props	Props, die an die Komponente von «aussen» übergeben werden
setup	Beinhaltet die Logik der Komponente
emits …	Weitere Objekte, hier nicht verwendet.

Wir halten es für wenig sinnvoll, hier auf alle technischen Änderungen im Detail einzugehen. Nachdem alle Komponenten in die notwendige Form gebracht wurden, musste Quasar noch leicht angepasst und die Webpack-Build-Pipeline aktualisiert werden.

Zu guter Letzt gelang es uns aber, alle oben genannten Packages auf den neuesten Stand zu bringen. Dadurch wurden im Vergleich zu vorher laut unserem Package-Manager NPM 8 kritische sowie 23 schwere Sicherheitslücken behoben:

```
x86 git:(main) npm install

up to date, audited 1279 packages in 1s

153 packages are looking for funding
  run `npm fund` for details

3 moderate severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Die drei verbleibenden Sicherheitslücken sind auf den «vue-prism-editor» [34] zurückzuführen, der seit mehr als neun Monaten nicht mehr aktualisiert wurde.

2.2. Google Lighthouse Tests

Der Google Lighthouse Test ist ein automatisiertes Open-Source Tool, das die Performance und Qualität einer Webseite evaluiert. Wir waren hier an der Performance unserer Webapplikation interessiert und wollten wissen, wie sich unser Score durch die Migration verändert hat.

Um einen möglichst fairen Vergleich zu ermöglichen, haben wir den Code direkt vor und direkt nach der Migration miteinander verglichen. Die Builds wurden lokal mit einem Apache Server ausgeliefert und via Chrome ausgewertet (Version: 102.0.5005.115)

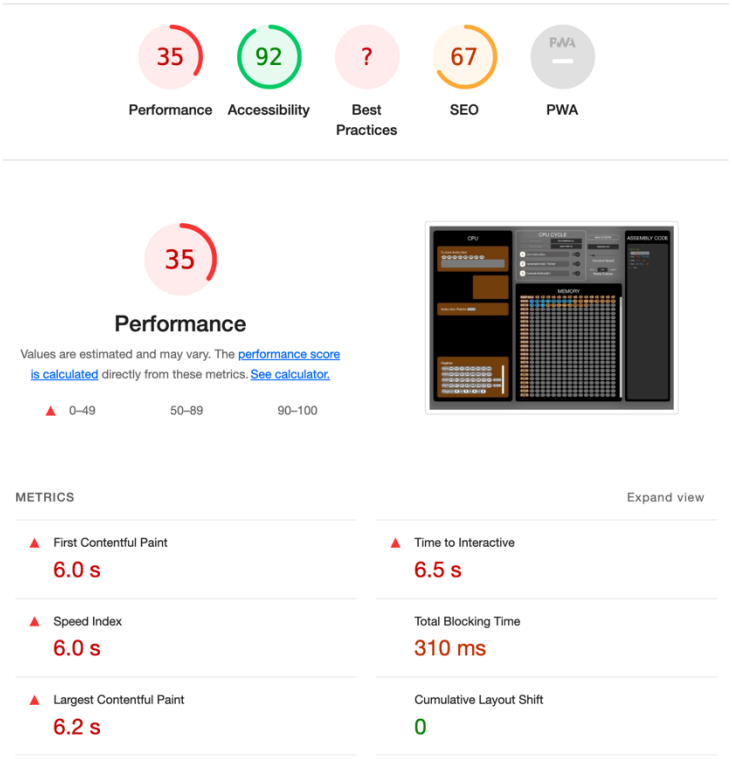

Abbildung 33 - Lighthouse Test vor der Migration

Der Test für den Vue 2-Build gibt der Webapplikation eine Performance-Note von 35 (von 100) - das ist unsere Ausgangslage.

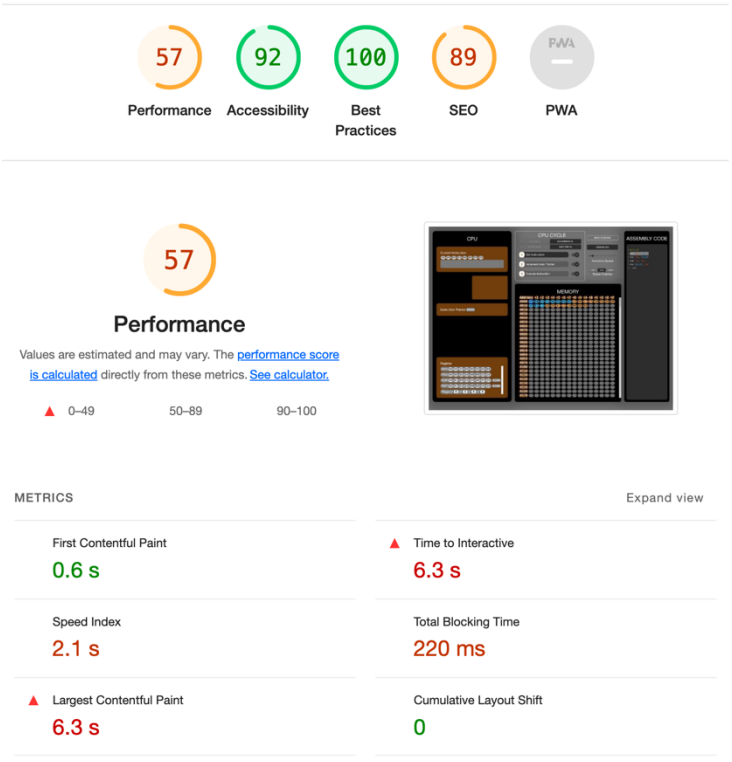

Abbildung 34 - Lighthouse Test nach der Migration

Die Vue 3 Version unserer Webapplikation besitzt eine bessere Performance-Note von 57. Die folgenden Werte haben sich verbessert:

Metrik	**Veränderung in Sekunden**
First Contentful Paint	-5.4s
Speed Index	-3.9s
Largest Contentful Paint	+0.1s
Time To Interactive	-0.2s
Total Blocking Time	-0.09s

Abschliessend kann das folgende Fazit gezogen werden. Die Applikation zeigt schneller Inhalte an und es ist früher möglich, mit dem Simulator zu interagieren. Für einen vollwertigen Aufbau der Nutzeroberfläche und das Initialisieren der gesamten Applikation vergeht fast gleich viel Zeit. Wir gewannen hier hauptsächlich an Reaktivität – die Zeit, die Nutzende darauf warten müssen, bis die Webseite Inhalte darstellt.

Weiterentwicklungspotential: Die Portierung von Vue 2 nach Vue 3 wurde, soweit möglich, ohne Veränderungen vorgenommen. in diesem Prozess wurde die «Byte.vue» Komponente mit Caching versehen [35]. Der Grossteil des Simulators besass kein Caching. Um eine möglichst schnelle Migration durchführen zu können, wurde das Caching nicht weiter ausgebaut. Falls die Performance weiter verbessert werden soll, sollte hier angesetzt werden und das Caching für weitere Komponenten aktiviert werden.

3. Nachteile

Wir waren in der Lage, die Applikation fast ohne jegliche Veränderung an der Funktionalität zu aktualisieren. Das einzige Feature, das wir nicht beibehalten konnten, war die Packbarkeit in eine einzige HTML-Datei. Früher wurde der Simulator als eine einzige Datei ausgeliefert, die alle Libraries, Bilder, etc. beinhaltete. Die drei Webpack-Bibliotheken, die für dieses Feature zuständig waren, wurden alle nicht auf Webpack 5 aktualisiert [36] und können deshalb nicht weiterverwendet werden. Neu sind es statt einer HTML Datei also drei Dateien:

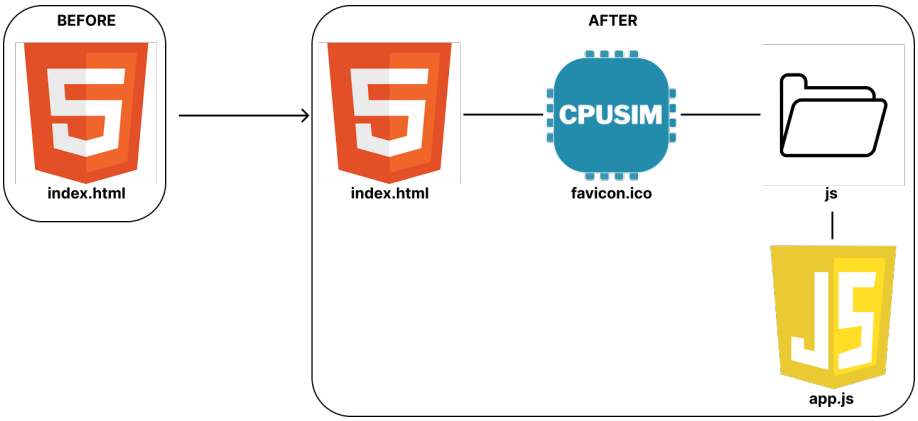

Abbildung 35 - Auslieferung der Web-Applikation zuvor und jetzt

Qualitätssicherung

1. Tests

Ein grosser Teil der Kernfunktionalität wird mit Testfällen abgedeckt. Wir verwenden dafür die von Vue bereitgestellten Vue Test-Utilities. Insgesamt stellen 220 Regressionstests sicher, dass bestehende Funktionalität nicht durch neue Features beeinträchtigt wird. Speziell für die Reverse Debugging-Komponente waren wir sehr dankbar um die Test-Bibliothek.
Der Grossteil der neugeschriebenen Testfälle befasst sich mit dem Expression-Parser.

2. SonarQube & Sicherheitslücke im Code Viewer

Um die Qualität unseres Codes laufend zu überprüfen, installierten wir ausserdem **SonarQube** [37] (Community Edition Version 9.3). Obwohl dessen Code-Analyse teilweise Schwierigkeiten hatte, die Eigenheiten der verwendeten Frameworks zu verstehen, wies es uns dennoch auf wichtige Versäumnisse in unserem Code hin. Die nennenswerteste davon war ein potenzieller XSS-Exploit, den wir versehentlich im neu überarbeiteten Code Viewer eingebaut hatten.

2.1. Ausgangslage

Aufgrund der bereits beschriebenen Ersetzung des Prism-Editors im Code Viewer durch eine selbstgebaute HTML-Table mussten wir das Prism Syntax Highlighting separat einbinden, um die Lesbarkeit des ASM-Codes aufrecht zu erhalten. Der Prism Highlighter akzeptiert Text als Input und gibt HTML-Tokens zurück, die den Text farblich kennzeichnen. Der Output ist also HTML-Code, den man direkt 1:1 darstellen kann. Dazu verwendet man in Vue die Direktive v-html.

2.2. Exploit

Das Problem dabei ist, dass dieser HTML-Code (teilweise) direkt vom User bestimmt werden kann. Was auch immer der User als ASM-Code eingibt, wird dann mit Tokens versetzt und anschliessend als HTML-Code im Simulator angezeigt. Dies ist ein typisches Beispiel für einen XSS-Exploit: der User gibt Schadcode als ASM-Code ein, der Tokenizer übernimmt diesen, wandelt ihn durch Tokens ab und gibt ihn dann im Code Viewer aus, wo er vom Browser direkt als HTML ausgeführt wird.

2.3. Proof of concept

Natürlich wollten wir testen, ob man auf diesem Weg tatsächlich Schadcode im Simulator ausführen könnte. Das erste Problem dabei ist, dass sich der Editor ausschliesslich nach der Eingabe eines gültigen ASM-Programms verlassen (und der Simulator damit starten) lässt. Beim Klick auf «Start Program» wird ein Syntax-Check auf dem eingegebenen Code durchgeführt. Ist dieser nicht erfolgreich, wirft der Editor eine Fehlermeldung und startet den Simulator nicht.
Allerdings hatten wir als Programmierer des Simulators den Vorteil, über ein genaues Verständnis der Applikation zu verfügen. Daher wussten wir, dass der Editor das ASM-Programm in base64 codiert und mittels der URL dem Simulator übergibt. Der Simulator überprüft diesen String nicht erneut auf Gültigkeit als ASM-Programm. Somit kann ein Angreifer den gewünschten Schadcode also einfach in base64 codiert direkt per Link dem Simulator übergeben. Damit lässt sich der Angriff auch einfach durch einen böswilligen Link verbreiten: der Linktext ist für das Auge harmlos, da der Schadcode codiert ist.
Wir möchten anmerken, dass wir diesen Angriffsweg zwar aufgrund unserer Vertrautheit mit der Applikation erkannten, es aber für einen geschulten Angreifer ebenso leicht zu entdecken wäre. Schliesslich ist die Applikation open-source. Aber selbst ohne den Source-Code ist die Kodierung in der Adresszeile des Browsers leicht zu erkennen.

Wir hatten es also geschafft, den Schadcode in den Simulator zu bringen, allerdings wurde er zu diesem Zeitpunkt nach wie vor als reiner Text angezeigt und noch nicht ausgeführt. Der Grund dafür lag darin, dass der Tokenizer die Eingabe des Nutzenden mit HTML-Tokens umgab, wodurch der Schadcode selbst als Text interpretiert wurde.

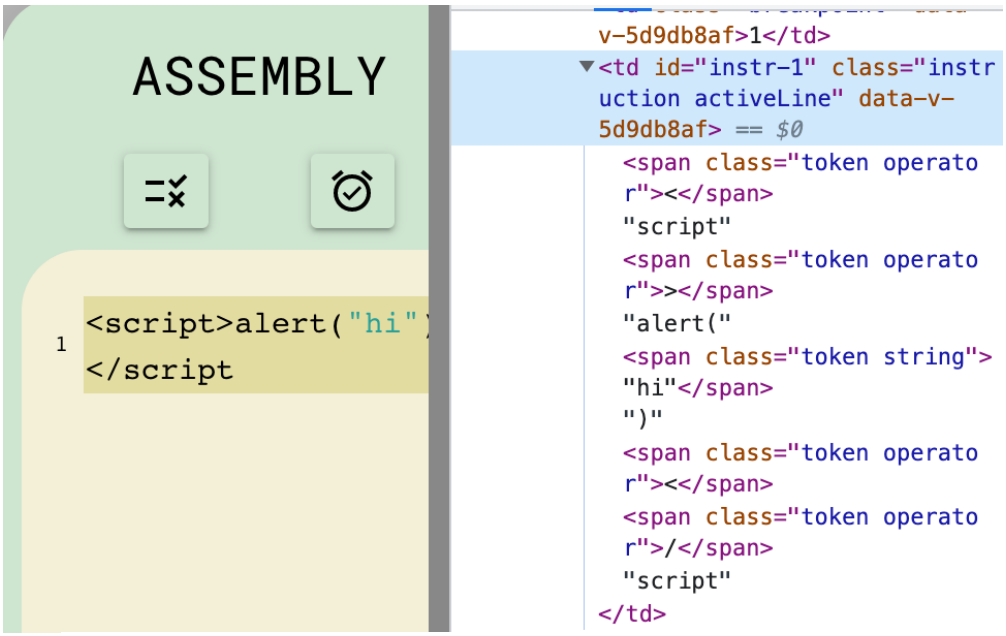

Abbildung 36 - Schadcode inklusive HTML-Interpretation durch Browser

Es war uns selbst nach Zuhilfenahme einiger Security-Experten aus dem Arbeitsumfeld eines der Autoren nicht möglich, diese Limitierung zu umgehen und Schadcode tatsächlich auszuführen. Allerdings war die darauf verwendete Zeit stark limitiert und keiner der Tester auf das Ausnutzen solcher Exploits spezialisiert. Alle Beteiligten waren sich daher einig, dass es vermutlich möglich wäre, durch cleveres Ausnützen der Browser-spezifischen Interpretation von fehlerhaftem HTML-Code auf irgendeine Art und Weise den Code zur Ausführung zu bringen.

2.4. Reparatur des Exploits

Daher entschlossen wir uns, dieses potenzielle Sicherheitsrisiko aus reiner Vorsicht zu reparieren. Da das Problem hauptsächlich auf dem Vorhandensein von Schadcode im User-Input beruht, macht eine Bereinigung (genannt «Sanitation») dieser Eingabe Sinn. Der Industriestandard hierfür ist die Verwendung der Library **DOMPurify** [38]. Der neue Programmablauf beim Laden des Simulators funktioniert also wie folgt: wir dekodieren den per URL übergebenen Textstring mit dem

(vermeintlichen) ASM-Programm und übergeben diesen String dann DOMPurify. Die Library entfernt alles, was JavaScript-Schadcode ähnelt und gibt uns einen bereinigten String zurück. Diesen übergeben wir anschliessend dem Tokenizer und können das Ergebnis direkt als HTML-Code im Code Viewer darstellen.
In unseren Tests funktioniert dieses Verfahren gut. Wann immer man nun versucht, Schadcode einzufügen, erhält man nur leere Zeilen. Korrekter ASM-Code war in unseren Tests nie betroffen.

Fazit

Insgesamt sind wir sehr zufrieden mit unserer Arbeit am Simulator. Es gelang uns, die anfangs sehr limitierte Code-Anzeige so auszubauen, dass Nutzende dadurch die vollständige Kontrolle über die Ausführung des Programms haben. Durch die Kombination aus Breakpoints, bedingten Breakpoints, Watchpoints und klickbaren Zeilen können Nutzende schnell an die interessanten Stellen ihrer Programme springen. Das Dialogfeld beim Definieren bedingter Breakpoints oder Watchpoints erlaubt dabei das Erstellen komplexer Haltebedingungen. Die Markierungen der aktuellen Zeile einerseits und den Instruktionen im Speicher andererseits machen es allen Benutzerinnen und Benutzern einfach, den aktuellen Zustand der Applikation im Verhältnis zum ASM-Programm nachzuvollziehen.

Durch unsere Migration haben wir die Applikation auf den neuesten Stand der Technik gebracht. Dadurch ist sie einfach erweiterbar und wartbar. Zahlreiche Sicherheitslücken, die während der letzten zwei Jahre bekannt wurden, sind nun behoben.

Beim Implementieren der obengenannten Features haben wir durch zahlreiche Tests sichergestellt, dass Qualität, Sicherheit und Performance nicht unter den Veränderungen leiden. Durch die Migration gelang es uns, die Performance der Applikation sogar noch zu verbessern.

Alles in allem haben wir das Gefühl, den Simulator in ein einfach verwendbares und ausgereiftes Produkt verwandelt zu haben. Wie bei Software-Projekten aber üblich, gibt es immer noch weitere Möglichkeiten zur Verbesserung des Bestehenden oder die Ergänzung durch neue Funktionalitäten. Wir haben uns bemüht, derartige Fälle in dieser Arbeit zu beschreiben, um zukünftigen Entwicklern die Arbeit zu erleichtern.

Wir möchten uns bei Prof. Stefan Richter für die ausgezeichnete Betreuung unserer beiden Arbeiten bedanken. Seine Zeit, Geduld und kreativen Denkanstösse haben wesentlich zu dieser Arbeit beigetragen.

Ausserdem möchten wir allen Studierenden danken, die uns Feedback zum Simulator gegeben haben und ihnen alles Gute für ihr weiteres Studium wünschen.

Wir hoffen, mit dieser Arbeit einen hilfreichen Beitrag zur Ausbildung künftiger Studierender geleistet zu haben.

Literaturverzeichnis

[1] „NASM Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.nasm.us

[2] „TypeScript Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.typescriptlang.org

[3] „Vue.js Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org

[4] „Quasar Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://quasar.dev

[5] „Unicorn Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.unicorn-engine.org

[6] „Capstone Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.capstone-engine.org

[7] „Ndisasm Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.csie.ntu.edu.tw/~comp03/nasm/nasmdoca.html

[8] „Visual Studio Code Debugger Protocol“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://microsoft.github.io/debug-adapter-protocol/

[9] „DeZog Github Page“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/maziac/DeZog

[10] „DeZog SetBreakpoint API“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: b/99349c2895e2f093850c45c56031554b3984f581/design/DeZogProtocol.md#cmd_set_breakpoints13

[11] „Prism.js Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://prismjs.com/index.html

[12] „NASM Instruction Manual“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://nasm.us/doc/nasmdoc0.html

[13] „VSCode Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://code.visualstudio.com

[14] „VSC Breakpoint Github“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/microsoft/vscode/blob/75a6ddc8626b831f01f91f9f60f27de3d5bb6f38/src/vs/workbench/api/common/extHostTypes.ts

[15] „JS Documentation: Set“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

[16] „Google Material Symbol Library“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://fonts.google.com/icons

[17] „Abstract Syntax Trees Wikipedia“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Abstract_syntax_tree

[18] „AST Explorer Project“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://astexplorer.net

[19] „Expression Parser NPM Library“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.npmjs.com/package/expressionparser

[20] „Expression Parser GitHub“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/dcollien/ExpressionParser

[21] „JavaScript Int Datentyp - Dokumentation“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER

[22] „JavaScript BigInt Datentyp - Dokumentation“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt
[23] „Vue Query Builder Repository“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/rtucek/vue-query-builder
[24] „jQuery QueryBuilder Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://querybuilder.js.org
[25] „Vue Class Component NPM Library“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.npmjs.com/package/vue-class-component
[26] „NPM Semantic Versioning“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://docs.npmjs.com/about-semantic-versioning
[27] „Webpack Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://webpack.js.org/
[28] „Vue Class Component Deprecation Github Issue #4744“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/vuejs/core/issues/4744
[29] „Vue Class Component Deprecation Github Issue #569“. Zugegriffen: 15. Juni 2022. [Online]. Verfügbar unter: https://github.com/vuejs/vue-class-component/issues/569
[30] „Vue Composition API“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org/guide/extras/composition-api-faq.html
[31] „Vue Options API“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org/guide/introduction.html#api-styles
[32] „Vue Typescript Guideline“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org/guide/typescript/overview.html
[33] „Vue Options API with Typescript“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org/guide/typescript/options-api.html
[34] „Prism NPM Library“.
[35] „Vue 3 Composition API Computed“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://vuejs.org/guide/essentials/computed.html
[36] „Webpack 5 Release“. Zugegriffen: 15. Juni 2022. [Online]. Verfügbar unter: https://webpack.js.org/blog/2020-10-10-webpack-5-release/
[37] „SonarQube Homepage“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://www.sonarqube.org
[38] „DOMPurify GitHub Page“. Zugegriffen: 16. Juni 2022. [Online]. Verfügbar unter: https://github.com/cure53/DOMPurify

Abbildungsverzeichnis