

**Bachelorarbeit****ARM Simulator, Interpreter und  
Debugger als Webanwendung**

Dominik Zangerl

Betreuer: Alexander Schlögl

Innsbruck, 29. September 2021

**Eidesstattliche Erklärung**

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

---

29.09.2021

Datum

---

Unterschrift

## Kurzfassung

Der ARM Simulator<sup>1</sup> stellt eine ARMv5 Entwicklungsumgebung als Webanwendung zur Verfügung. Die Anwendung verwendet einen Parser basierend auf einer Parsing Expression Grammatik, mit dem die Benutzereingabe schnell und effizient analysiert werden kann. Der Code von Benutzer:innen wird anschließend in einen simulierten Hauptspeicher geschrieben und kann mit der Code Execution Engine, die auch als Debugger dient, ausgeführt werden. Mit dem Debugger kann dann der Code Zeile für Zeile oder bis zu bestimmten Breakpoints ausgeführt werden. Während der gesamten Ausführung wird dabei der Zustand der Register, des Statusregisters und des Hauptspeichers angezeigt. Der Inhalt der Register und des Hauptspeichers kann jederzeit von Benutzer:innen verändert werden. Diese Funktionen vereinfachen die Fehlersuche und das Debugging der Assembler Programme. Der Simulator ist in TypeScript geschrieben und benutzt das Webframework React als Frontend. React funktioniert in jedem modernen Browser und Benutzer:innen können, ohne Installation von zusätzlichen Programmen oder Tools, ihren ARMv5 Code direkt in ihrem Webbrowser ausführen und analysieren.

---

<sup>1</sup>Der Code für den Simulator ist auf <https://github.com/Koro95/ARM-Simulator> zu finden.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Theorie</b>	<b>2</b>
2.1	ARMv5 . . . . .	2
2.1.1	Architektur . . . . .	2
2.1.2	Bedingungsfeld . . . . .	5
2.1.3	Datenverarbeitende Instruktionen . . . . .	6
2.1.3.1	Adressierungsarten . . . . .	7
2.1.4	Instruktionen für Multiplikation . . . . .	8
2.1.5	Instruktionen für Sprünge . . . . .	9
2.1.6	Lade– und Speicherinstruktionen . . . . .	9
2.1.6.1	Adressierungsarten . . . . .	10
2.1.7	Lade– und Speicherinstruktionen für mehrere Register . . . . .	11
2.1.7.1	Adressierungsarten . . . . .	11
2.2	Parsing Expression Grammatik und tsPEG . . . . .	13
<b>3</b>	<b>Implementation</b>	<b>15</b>
3.1	Instruktionen und Operanden . . . . .	15
3.2	Übersicht . . . . .	18
3.3	Parser . . . . .	19
3.4	Hauptspeicher . . . . .	22
3.5	Code Execution Engine . . . . .	23
3.6	Benutzeroberfläche . . . . .	26
<b>4</b>	<b>Evaluation</b>	<b>29</b>
<b>5</b>	<b>Zusammenfassung</b>	<b>31</b>
<b>A</b>	<b>Parsing Expression Grammatik</b>	<b>33</b>
<b>B</b>	<b>Benchmark Code</b>	<b>36</b>
B.1	Division . . . . .	36
B.2	Binomialkoeffizient . . . . .	37

# 1 Einleitung

Das Ziel meiner Bachelorarbeit ist es eine Webanwendung zu entwickeln, mit der die ARMv5 Entwicklungsumgebung simuliert wird. ARMv5 [2] wird im ersten Semester als Beispiel für eine Befehlssatzarchitektur unterrichtet. Studierende sollen ihre eigenen Programme in Assembler [8] schreiben und diese dann auf einer ARMv5 Architektur ausführen. Diese Entwicklungsumgebung wird zurzeit mit verschiedenen Linux-Programmen simuliert. Die GNU Toolchain für die ARM Cortex-A Architektur [1] wird für das Kompilieren und Linken der Assembler-Dateien verwendet. Das kompilierte ARM-Programm läuft dann nicht auf der Architektur des Hostrechners, sondern wird mit Hilfe des QEMU User-Space-Emulators [13] ausgeführt. Dieser Prozess kann vereinfacht werden, indem man die Toolchain in einer virtuellen Maschine oder dem Windows Subsystem for Linux [11] installiert und sich für die Befehlskette ein Skript schreibt.

Das größere Problem bei dieser Toolchain ist die Fehlersuche und das Debugging des Programms. Den Fehler auf eine bestimmte Instruktion oder ein Register zurückzuführen nimmt oft die größte Zeit in Anspruch. Der ARM-Emulator kann zusammen mit dem GNU Debugger [12] verwendet werden, welcher auch die Inhalte der Register anzeigen kann. Dies bedeutet jedoch häufig einen großen Zeitaufwand um alles aufzusetzen. Auch das Arbeiten mit Debuggern, besonders auf der Kommandozeile, könnte vielen noch nicht geläufig sein.

An dieser Stelle greift dieses Bachelorprojekt ein und versucht die ARMv5 Entwicklungsumgebung inklusive Debugging mit einer Webanwendung zu simulieren. Benutzer:innen schreiben den ARM-Code direkt in die Webanwendung, welcher dann auf einer simulierten CPU und simuliertem Hauptspeicher direkt im Browser ausgeführt wird. Die Inhalte der Register, des Stacks und Teile des Hauptspeichers werden dauerhaft angezeigt und helfen Benutzer:innen bei der Fehlerbehebung, da sie sofort sehen, an welcher Stelle ein ungewünschter Wert in ein Register geschrieben wird. Zusammen mit den Funktionen eines Debuggers, wie zeilenweise Abarbeitung des Codes oder setzen von Breakpoints, wird den Studierenden die zeitaufwändigste Arbeit abgenommen und sie können sich auf den wichtigen Teil konzentrieren, nämlich das Schreiben und Verstehen von ARM-Assembler Code.

In Abschnitt 2 werden zuerst die ARM Architektur, die implementierten Instruktionen und Parsing Expression Grammatiken am Beispiel von tsPEG [5] erklärt. Abschnitt 3 beschreibt meine konkrete Implementation, die einzelnen Komponenten des Simulators und die Benutzeroberfläche. In Abschnitt 4 folgt eine kurze Evaluation der Performance des Simulator anhand von zwei Beispielen. Abschnitt 5 schließt mit einer kurzen Zusammenfassung ab. Im Appendix sind noch die Grammatik in Abschnitt A und die für die Evaluation verwendeten Programme in den Abschnitten B.1 und B.2 zu finden.

## 2 Theorie

### 2.1 ARMv5

In den nächsten Abschnitten beschreibe ich alle für die Implementation des ARM Simulators benötigten Teile der ARM Architektur. Bei den Spezifikationen orientiere ich mich dabei am ARM Referenzhandbuch – Ausgabe I [2].

#### 2.1.1 Architektur
















Bei der ARM Architektur handelt es sich um einen Rechner mit reduziertem Befehlssatz (RISC – Reduced Instruction Set Computer). Sie zeichnet sich hauptsächlich durch eine Load/Store–Architektur aus, bei der datenverarbeitende Instruktionen nur mit den Inhalten der Register arbeiten und nicht direkt mit den Daten im Hauptspeicher [4]. Dazu gibt es eine große einheitliche Register–Datei und einfache Adressierungsarten für das Laden und Speichern von Daten, bei denen die Speicheradressen nur aus den Registern oder Feldern der Instruktion geladen werden. Außerdem haben alle Instruktionen eine einheitliche Form und Länge um das Kodieren/Dekodieren zu vereinfachen [2][4].

Darüber hinaus bietet die ARM Architektur noch einige Erweiterungen zu einer normalen RISC Architektur [2]:






- Die meisten datenverarbeitenden Instruktionen haben Zugriff auf die arithmetisch-logische Einheit (ALU – Arithmetic Logic Unit) und den Barrel–Shifter.
- Die Adressierungsarten bieten Möglichkeiten die Adresse automatisch zu inkrementieren/dekrementieren.
- Um den Datendurchsatz zu erhöhen, gibt es Instruktionen um mehrere Register zu laden und zu speichern.
- Die Ausführung fast aller Instruktionen kann mittels Bedingungen bestimmt werden.

Im Gegensatz dazu haben Rechner mit komplexem Befehlssatz (CISC – Complex Instruction Set Computer) Instruktionen mit unterschiedlicher Länge, mehr Adressierungsarten und können direkt auf dem Hauptspeicher arbeiten [4]. Mit Berücksichtigung von Leistung und Energieeffizienz haben sich besonders früher in den 1980er Jahren RISC Architekturen auf mobile Geräten und eingebettete Systeme konzentriert, während CISC Architekturen hauptsächlich in Desktop–Computern und Servern eingesetzt wurden. In aktuelleren Studien von Blem et al. [4] wurden die Unterschiede der Befehlssätze bei neueren Prozessoren erneut mit Hinsicht auf diese Eigenschaften untersucht und Unterschiede auf Eigenschaften unabhängig vom Befehlssatz, wie Design der Prozessorkerne zurückgeführt.

ARM verfügt über 31 Universal–Register mit einer Breite von 32 Bit. Es sind immer nur 16 dieser Register sichtbar. Welche Register sichtbar sind hängt vom derzeitigen Ausführungsmodus des Prozessors ab. Dazu gibt es noch 6 Statusregister mit 32 Bit Breite, von denen nicht immer alle Bits verwendet werden. Abbildung 1 zeigt die sichtbaren Register je nach Ausführungsmodus. In der restlichen Arbeit und für den Simulator wird lediglich der User–Modus berücksichtigt, da alle betrachteten und implementierten Instruktionen in diesem Modus arbeiten. Im User–Modus kann nur unprivilegierter Code ausgeführt werden, man kann nur über spezielle Instruktionen (z.B. Software–Interrupts, siehe Sektion []) in einen anderen Ausführungsmodus wechseln und es besteht nur eingeschränkter Zugriff auf Speicher und Koprozessoren. Register R0 bis

Ausführungsmodi						
<div> <div>Privilegierte Ausführungsmodi</div> <div>Ausnahmenmodi</div> </div>						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
R13	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
R14	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq


 zeigt an, dass normale Register aus User- und Systemmodus durch alternative Register spezifisch für den jeweiligen Ausnahmemodus ersetzt wurden

Abbildung 1: Verfügbare Register je nach Ausführungsmodus [2]

R7 weisen in jedem Ausführungsmodus auf die gleichen 32 Bit breiten physikalischen Register. Register R8 bis R14 können in privilegierten Ausführungsmodi auf unterschiedliche physikalische Register zeigen (siehe Abbildung 1). Einige Register übernehmen dabei spezielle Funktionen [2]:

- R13 wird als Stapelzeiger (Stack Pointer – SP) verwendet und zeigt auf die Speicheradresse des momentanen Stapeleintrags. Die Adresse in diesem Register wird deshalb oft als Startadresse bei Lade– und Speicherinstruktionen verwendet.
- R14 ist das Link–Register (LR), in dem bei Sprungoperationen die Adresse der nächsten Instruktion nach Beendigung der Subroutine gespeichert wird. Am Ende der Subroutine wird dazu der Inhalt des Link–Registers in das Register für den Befehlszähler geladen.
- R15 ist das Register für den Befehlszähler (Program Counter – PC). Dieses Register

wird von allen Ausführungsmodi geteilt. Der Befehlszähler zeigt auf die nächste Instruktion die ausgeführt wird.

31	30	29	28	27	26	25	24	23	20	19	16	15	10	9	8	7	6	5	4	0	
N	Z	C	V	Q	Res	J	RESERVED			GE[3:0]		RESERVED			E	A	I	F	T	M[4:0]	

Abbildung 2: Inhalt des Status-Registers [2]

Das Status-Register (Current Program Status Register – CPSR) wird ebenfalls von allen Ausführungsmodi geteilt. Der Inhalt der einzelnen Bits ist in Abbildung 2 dargestellt. Es beinhaltet die Status-Flags (NZCV), Bits um Interrupts zu deaktivieren (A-, I- und F-Bit), Bits zum wechseln des Instruktions-Sets (T-Bit für Thumb-Instruktionen und J-Bit für Java Hardwarebeschleunigung) und Bits mit Informationen über den aktuellen Ausführungsmodus (M[4:0]-Bits). Die weiteren Bits sind nur für bestimmte Varianten (E-Bit bei Instruktionen für digitale Signalverarbeitung), werden erst ab ARMv6 verwendet (E-Bit um die Lade- und Speicher Endianness zu bestimmen, GE[3:0]-Bits sind größer-oder-gleich-Flags für die einzelnen Bytes oder Halbwörter bei SIMD Instruktionen) oder sind reserviert (Bits [26:25], [23:20] und [15:10]) [2].

Die Status-Flags in den höchsten 4 Bits können von Instruktionen im User-Modus verändert werden. Sie werden bei Vergleichsoperationen und Instruktionen, bei denen das S-Bit gesetzt wird, geupdated [2]:

- N** Negativ – Wird auf Bit 31 vom Ergebnis der Instruktion gesetzt. Entspricht dem Vorzeichen einer Zahl im Zweierkomplement → N = 1 bei einer negativen Zahl und N = 0 bei einer positiven Zahl oder Null.
- Z** Null (Zero) – Wird auf 1 gesetzt, falls das Ergebnis Null ist, andernfalls auf 0.
- C** Übertrag (Carry) – Hier wird zwischen 4 verschiedenen Fällen unterschieden:
  1. Addition: C wird auf 1 gesetzt, falls es einen vorzeichenlosen Überlauf gibt und ein Übertrag entsteht (Ergebnis größer als 32 Bit), andernfalls auf 0.
  2. Subtraktion: C wird auf 0 gesetzt, falls es einen vorzeichenlosen Unterlauf gibt und ein Borrow entsteht (Subtrahend < Minuend), andernfalls auf 1.
  3. Instruktionen mit Barrel-Shifter: C wird auf das zuletzt raus geschobene Bit gesetzt.
  4. Restliche Instruktionen: C wird nicht verändert.
- V** Überlauf (Overflow) – Hier wird wiederum zwischen 2 Fällen unterschieden:
  1. Addition/Subtraktion: V wird auf 1 gesetzt, falls es einen arithmetischen Überlauf gibt, andernfalls auf 0.
  2. Restliche Instruktionen: V wird nicht verändert

ARM unterstützt 3 verschiedene Datentypen. Wörter mit einer Breite von 32 Bit, Halbwörter (16 Bit) und Bytes (8 Bit). Die meisten Operationen arbeiten mit Wörtern, Lade- und Speicheroperationen können auch mit Halbwörtern und Bytes arbeiten (Laden: Halbwörter/Bytes werden mit Nullen erweitert, Speichern: nur die 8/16 untersten Bits werden gespeichert). Adressen im Hauptspeicher haben ebenfalls 32 Bit und eine Speicherausrichtung von 4 Bit (alle Adressen sind ein Vielfaches von 4) [2].



### 2.1.2 Bedingungsfeld

Kodierung [31:28]	Mnemonik	Bedeutung	Status-Flags
0000	EQ	Gleichheit (Equal)	Z=1
0001	NE	Ungleichheit (Unequal)	Z=0
0010	CS/HS	Carry-Bit gesetzt (Carry set)/ Vorzeichenlos größer oder gleich (unsigned higher or same)	C=1
0011	CC/LO	Carry-Bit nicht gesetzt (Carry clear)/ Vorzeichenlos kleiner (unsigned lower)	C=0
0100	MI	Negativ (Minus)	N=1
0101	PL	Positiv (Plus)	N=0
0110	VS	Überlauf (Overflow/V set)	V=1
0111	VC	Kein Überlauf (No Overflow/V clear)	V=0
1000	HI	Vorzeichenlos größer (Unsigned higher)	C=1, Z=0
1001	LS	Vorzeichenlos kleiner oder gleich (Unsigned lower or same)	C=0, Z=1
1010	GE	Größer oder gleich mit Vorzeichen (Signed greater than or equal)	N=1, V=1 oder N=0, V=0 (N == V)
1011	LT	Kleiner mit Vorzeichen (Signed less than)	N=1, V=0 oder N=0, V=1 (N != V)
1100	GT	Größer mit Vorzeichen (Signed greater than)	Z=0, N=1, V=1 oder Z=0, N=0, V=0 (Z == 0, N == V)
1101	LE	Kleiner oder gleich mit Vorzeichen (Signed less than or equal)	Z=1, N=1, V=0 oder Z=1, N=0, V=1 (Z == 1, N != V)
1110	AL	Immer (Always)	
1111		Reserviert für bedingungslose Ausführung	

Tabelle 1: Bedingungen für die Ausführung von Instruktionen [2]

Wie in Abschnitt 2.1.1 beschrieben, können fast alle Instruktionen unter bestimmten Bedingungen ausgeführt werden. Diese Bedingungen sind an die NZCV Status-Flags geknüpft. Ist die Bedingung erfüllt, wird die Instruktion normal ausgeführt, ist sie nicht erfüllt, wird die Instruktion übersprungen und der Befehlszähler erhöht. Bei der Kodierung einer Instruktion nimmt die Bedingung dabei immer die höchsten 4 Bits [31:28] ein, wie in Abbildung 3 zu



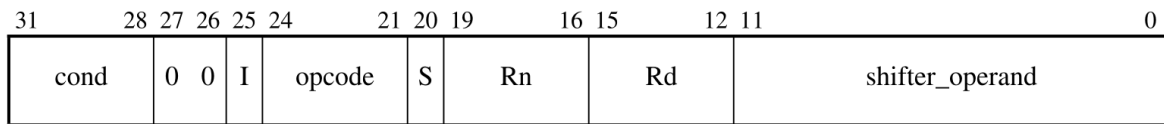


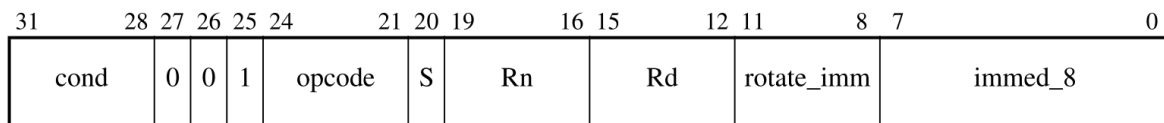
Abbildung 4: Kodierung von datenverarbeitenden Instruktionen [2]

Unterscheidung zwischen einem Register ( $I = 0$ ) und einem Immediate-Wert ( $I = 1$ ) im Shifter-Operanden verwendet. Danach folgt die Kodierung der Instruktion laut Tabelle 2. Das S-Bit an Stelle 20 gibt an, ob die Status-Flags nach der Instruktion aktualisiert werden sollen. Schließlich folgen der erste Quelloperand  $Rd$  (immer ein Register), das Zielregister  $Rn$  und der zweite Quelloperand (Shifter-Operand, siehe 2.1.3.1) [2].

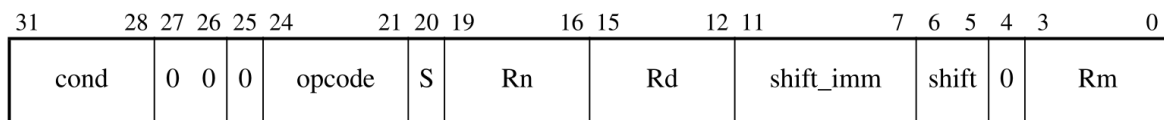
Nicht alle dieser Instruktionen berücksichtigen dabei jeden Operanden. In Tabelle 2 ist die jeweilige Aktion in der letzten Spalte gelistet und welche Operanden sie dafür benutzt. Die Test- und Vergleichsinstruktionen (TST, TEQ, CMP, CMN) nehmen nur die 2 Quelloperanden, berechnen das Ergebnis und aktualisieren die Status-Flags, ohne dabei das Ergebnis in ein Zielregister zu schreiben. Die Kopieroperationen (MOV, MVN) verwenden nur das Zielregister und das zweite Quellregister, da der Shifter-Operand flexibler ist als nur ein Register [2].

### 2.1.3.1 Adressierungsarten

#### 32 Bit Immediate-Wert



#### Verschiebeoperation mit Immediate-Wert



#### Verschiebeoperation mit Register

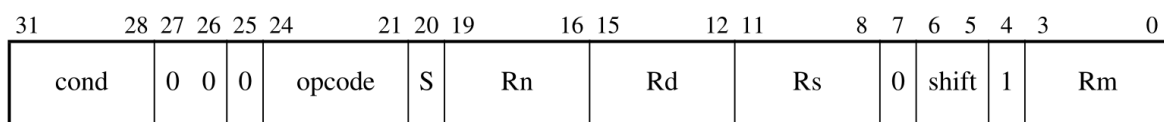


Abbildung 5: Adressierungsarten bei datenverarbeitenden Instruktionen [2]

Abbildung 5 zeigt die unterschiedlichen Adressierungsarten für datenverarbeitende Instruktionen. Der Unterschied besteht im zweiten Quelloperanden, dem sogenannten Shifter-Operand. Für diesen zweiten Operanden stehen 12 Bit zur Verfügung. Die erste Variante ist ein 8 Bit Immediate-Wert mit einer Rotation. Das heißt für die Rotation bleiben 4 Bit übrig, was 16 unterschiedliche Werte zulässt. Um die gesamten 32 Bit Breite eines Registers abzudecken, wird der Wert der Rotation dafür verdoppelt. Die 4 Bit Rotation kann also alle

geraden Werte von 0–30 annehmen. Dies bedeutet aber auch, dass dieser Operand nicht jeden Wert repräsentieren kann. Beliebige 32 Bit Werte können nur aus Registern geladen werden. Außerdem wird bei dieser Variante das I–Bit an Stelle 25 gesetzt [2].

Die andere Möglichkeit ist eine Verschiebeoperation mit einem Immediate–Wert oder Register. Das Register  $Rm$  in Bits [3:0] wird mit einem der 5 Shift-Typen (Logische Linksverschiebung/Arithmetische Linksverschiebung → LSL/ASL, Logische Rechtsverschiebung → LSR, Arithmetische Rechtsverschiebung → ASR, Rechtsrotation → ROR, Erweiterte Rechtsrotation um 1 Bit → RRX) in Abbildung 6 verschoben bzw. rotiert. Den Wert für den Immediate–Shift findet man in Bits [11:7] und für den Register–Shift in Bits [11:8]. Bit 4 wird zur Unterscheidung der beiden Varianten verwendet [2].

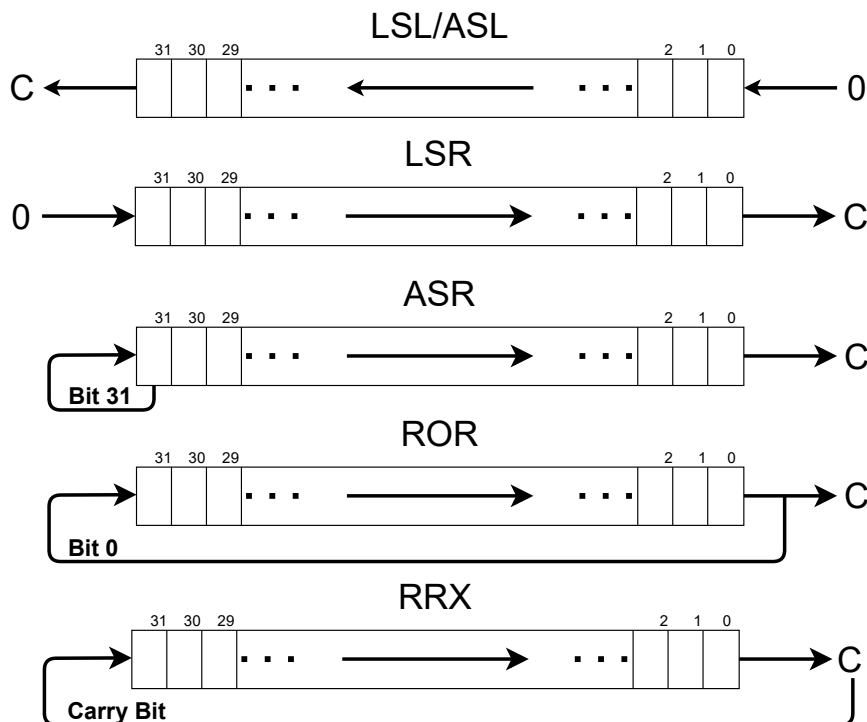


Abbildung 6: Visualisierung der verschiedenen Shift–Typen, inspiriert von [8]

#### 2.1.4 Instruktionen für Multiplikation

In ARM gibt es mehrere verschiedene Klassen von Multiplikation je nach Datentyp. Bei der normalen Multiplikation von 32 Bit Wörtern werden beim Ergebnis nur die untersten 32 Bit in einem Register gespeichert. Es gibt auch die lange Multiplikation, bei der 64 Bit des Ergebnisses aufgeteilt in 2 Register gespeichert werden. Außerdem gibt es Instruktionen für die Multiplikation von Halbwörtern und die Multiplikation von einem Wort mit einem Halbwort [2]. Im Weiteren und beim Simulator wird nur die normale Multiplikation betrachtet.

Die normale Multiplikation hat 2 verschiedene Instruktionen. Abbildung 7 zeigt die Multiplikation von Quellregister  $Rm$  mit Quellregister  $Rs$  und speichert die unteren 32 Bits vom Ergebnis in das Zielregister  $Rd$ . SBZ (Should-Be-Zero) in Bits [15:12] bedeutet, dass Software nur Nullen in diese Felder schreiben soll, da sonst ein unberechenbares Ergebnis entsteht [2].

Eine weitere Instruktion mit normaler Multiplikation ist MLA (Multiply Accumulate), bei der nach der Multiplikation der 2 Quellregister noch das 3. Quellregister  $Rn$  auf das Produkt

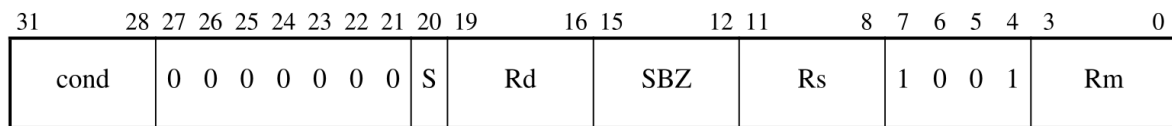


Abbildung 7: Kodierung der MUL Instruktion [2]

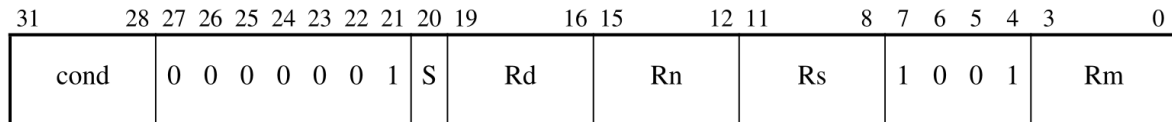


Abbildung 8: Kodierung der MLA Instruktion [2]

aufaddiert wird. Dafür werden die unbenutzten Bits [15:12] der vorherigen Instruktion für das 3. Quellregister genutzt. Bei beiden Instruktionen können durch Setzen des S-Bits die N- und Z-Flag aktualisiert werden. Die Flags für Carry und Überlauf werden dabei nicht verändert [2].

### 2.1.5 Instruktionen für Sprünge

Sprungoperationen verursachen eine Verzweigung in der Ausführung des Codes zu einer bestimmten Zieladresse. Diese Zieladresse wird in Form eines Labels angegeben, welches auf bestimmte Stellen im Hauptspeicher zeigt. Die Verzweigung wird hervorgerufen, indem die Zieladresse des Labels in das Register für den Befehlszähler geladen wird. Es gibt wieder unterschiedliche Klassen von Verzweigungen, die noch zusätzliche Effekte hervorrufen. In dieser Arbeit werden nur die normale Verzweigung und die Verzweigung mit Hinterlegung der Rücksprungadresse betrachtet. Zusätzlich würde es noch Sprunginstruktionen geben, die nach dem Sprung zu den in Abschnitt 2.1.1 erwähnten alternativen Befehlssätzen wechseln (z.B. BX – Branch and Exchange, wechselt zum Befehlssatz für Thumb-Instruktionen) [2].

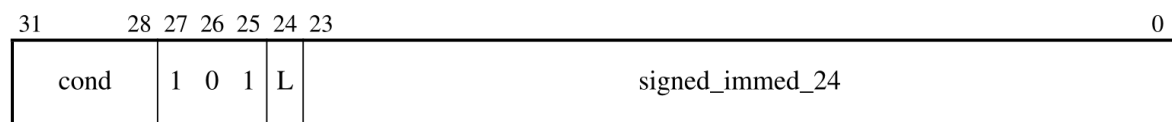


Abbildung 9: Kodierung der Instruktionen B und BL [2]

Abbildung 9 zeigt die Kodierung für die normalen Sprunginstruktion B (Branch). Ist das L-Bit an Stelle 24 gesetzt, handelt es sich um eine BL (Branch and Link) Instruktion, bei der die Rücksprungadresse (Adresse der Sprunginstruktion + 4) im Link-Register hinterlegt wird. Für den Rücksprung kann man dann einfach diese Adresse zurück in das Register für den Befehlszähler kopieren (MOV *pc, lr*). Für den Adressabstand (*signed\_immed\_24*) stehen 24 Bit zur Verfügung. Um diese Bits zu berechnen wird die Basisadresse (Adresse der Instruktion + 8) von der Zieladresse subtrahiert und *signed\_immed\_24* wird auf Bits [25:2] des Ergebnisses gesetzt. Mit diesem Adressabstand von 24 Bit und einer Speicherausrichtung, bei der Adressen Vielfache von 4 sind, lassen sich damit Sprünge von  $\pm 32\text{MB}$  realisieren [2].

### 2.1.6 Lade- und Speicherinstruktionen

Abbildung 10 zeigt die Kodierung von Lade- und Speicherinstruktionen. Eine Ladeinstruktion lädt den Inhalt der Adresse im Basisregister *Rn* in das Zielregister *Rd* und eine Spei-

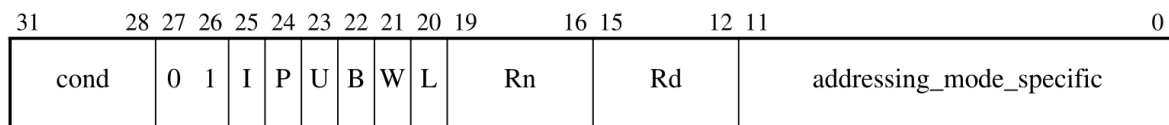


Abbildung 10: Kodierung der LDR/STR Instruktionen für Wörter und Bytes [2]

cherinstruktion speichert den Inhalt des Zielregisters *Rd* an die Adresse im Basisregister. In *addressing\_mode\_specific* kann außerdem noch ein optionaler Offset zum Basisregister angegeben werden. Dieser Offset kann wiederum ein Register, ein Immediate-Wert oder ein Shifter-Operand sein. Die verschiedenen Adressierungsarten werden in Abschnitt 2.1.6.1 besprochen. Die restlichen Bits sind Optionen um die Adressierungsarten unterscheiden zu können. Das I-Bit an Stelle 25 gibt an, ob der Offset ein Immediate-Wert ist, das P-Bit an Stelle 24 gibt an, ob es sich um die pre-indexed ( $P = 1$ ) oder post-indexed ( $P = 0$ ) Adressierungsart handelt, das U-Bit an Stelle 23 gibt an, ob der Offset addiert ( $U = 1$ ) oder subtrahiert ( $U = 0$ ) wird, das B-Bit an Stelle 22 gibt an, ob es sich um ein Byte ( $B = 1$ ) oder Wort ( $B = 0$ ) handelt, das W-Bit an Stelle 21 gibt an, ob die aktualisierte Adresse mit Offset bei pre-indexed Adressierung zurück in das Basisregister geschrieben wird und das L-Bit an Stelle 20 unterscheidet zwischen einer Lade ( $L = 1$ ) und Speicheroperation ( $L = 0$ ) [2].

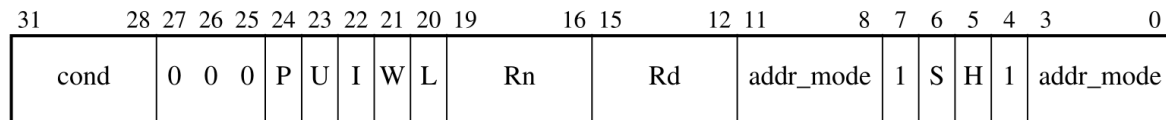


Abbildung 11: Kodierung der LDR/STR Instruktionen für Halbwörter und Laden von Bytes/Halbwörtern mit Vorzeichen [2]

Außerdem gibt es noch Instruktionen zum Laden/Speichern von Halbwörtern und das Laden von Bytes/Halbwörtern mit Vorzeichen. Beim Laden mit Vorzeichen wird das höchste Bit des Bytes/Halbworts auf die restlichen 32 Bit des Registers erweitert. Hier gibt es noch das S-Bit (Signed) an Stelle 6 für Vorzeichen und das H-Bit (Halfword) um Bytes von Halbwörtern zu unterscheiden. Die Adressierungsart ist auf Bits [11:8] und [3:0] aufgeteilt. Daher können nur Register in Bits [3:0] oder 8 Bit Immediate-Werte, auf beide Bit-Bereiche aufgeteilt, spezifiziert werden [2].

### 2.1.6.1 Adressierungsarten

Abbildung 12 zeigt die zwei Adressierungsarten bei Lade- und Speicherinstruktionen. Bei pre-indexed Adressierung wird der Offset innerhalb der eckigen Klammern angegeben und bei post-indexed Adressierung wird der Offset nach den eckigen Klammern angegeben. Der Unterschied befindet sich in der Adressierungsreihenfolge [2]:

- Pre-indexed**
1. Für die Adresse wird der Offset vor (pre) der Instruktion auf das Basisregister aufaddiert bzw. davon subtrahiert.
  2. Die Lade- oder Speicherinstruktion wird mit der berechneten Adresse ausgeführt.
  3. (Optional) Falls das W-Bit gesetzt ist, wird die berechnete Adresse zurück in das Basisregister geschrieben.

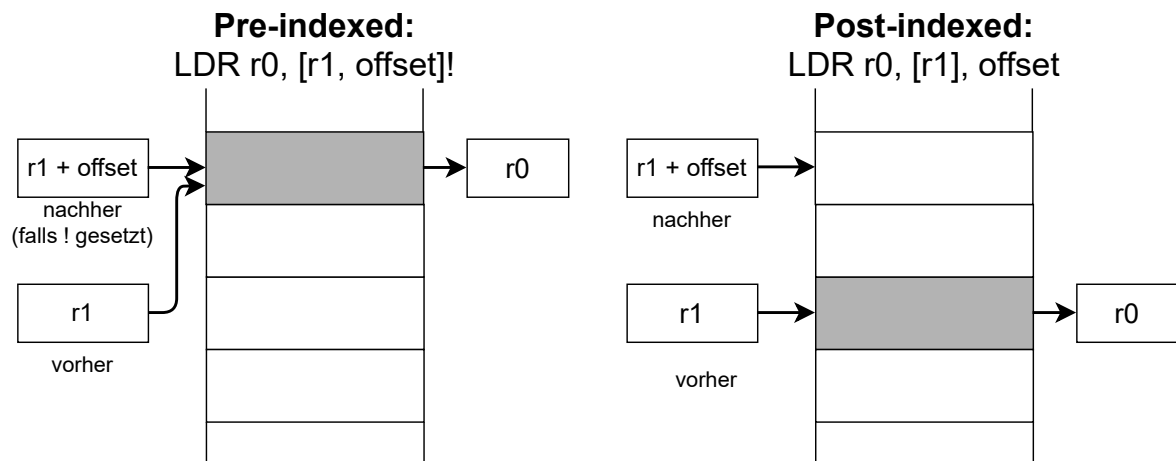


Abbildung 12: Adressierungsarten bei Speicher- und Ladeinstruktionen, inspiriert von [8]

- Post-indexed**
1. Die Lade- oder Speicherinstruction wird mit der Adresse im Basisregister ausgeführt.
  2. Der Offset wird nach (post) der Instruction auf das Basisregister addiert bzw. davon subtrahiert und zurück in das Basisregister geschrieben.

### 2.1.7 Lade- und Speicherinstructionen für mehrere Register

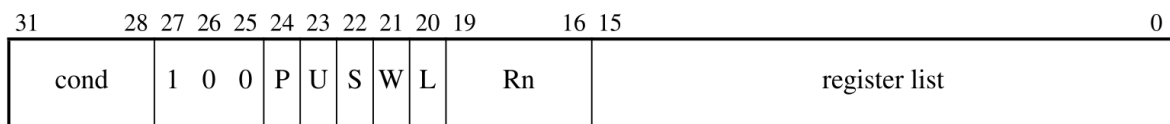


Abbildung 13: Kodierung der LDM/STM Instructionen [2]

Um Laden und Speichern effizienter zu machen, gibt es auch Instructionen bei denen man gleich mehrere Register spezifizieren kann. Abbildung 13 zeigt die Kodierung dieser Instructionen. Es gibt ein Quellregister  $Rn$  mit der Basisadresse und eine Liste von Registern ( $r0-r15$ ) in Bits [15:0]. Im Code können die einzelnen Register oder Register-Bereiche in eckigen Klammern angegeben werden (z.B.  $\{r0-r5, r8, sp, lr\}$ ). Die anderen Bits sind wieder für die Adressierungsarten in Abschnitt 2.1.7.1. Das P-Bit and Stelle 24 gibt an, ob die Adresse im Basisregister inkludiert wird ( $P = 0$ ), oder erst die nächsthöhere bzw. niedrigere Adresse betrachtet wird ( $P = 1$ ), das U-Bit (Upwards) and Stelle 23 gibt an, ob die Adresse erhöht ( $U = 1$ ) oder verringert ( $U = 0$ ) wird, das S-Bit an Stelle 22 ist 0 im User-Modus, das W-Bit an Stelle 21 gibt an, ob die aktualisierte Adresse zurück in das Basisregister geschrieben wird und das L-Bit an Stelle 20 unterscheidet zwischen einer Lade ( $L = 1$ ) und Speicheroperation ( $L = 0$ ) [2].

#### 2.1.7.1 Adressierungsarten

Bei Instructionen, die mehrere Register laden, muss sich natürlich auch die Zieladresse für die Lade- oder Speicherinstruction ändern. Dies geschieht, indem nach jeder Operation die Adresse um 4 erhöht bzw. verringert wird (Speicherausrichtung von 4 Bit). Mit den

Options-Bits aus dem vorherigen Abschnitt ergeben sich damit 4 verschiedene Adressierungsarten [2]:

- IA** Increment After ( $P = 0, U = 1$ ) – Der Wert im Basisregister wird als erste Adresse hergenommen und nach jeder Operation um 4 erhöht.
- IB** Increment Before ( $P = 1, U = 1$ ) – Die erste Adresse ist der Wert im Basisregister + 4 und wird vor jeder Operation um 4 erhöht.
- DA** Decrement After ( $P = 0, U = 0$ ) – Der Wert im Basisregister wird als erste Adresse hergenommen und nach jeder Operation um 4 verringert.
- DB** Decrement Before ( $P = 1, U = 0$ ) – Die erste Adresse ist der Wert im Basisregister – 4 und wird vor jeder Operation um 4 verringert.

Da Laden und Speichern von mehreren Registern oft mit einem Stack zusammen verwendet wird, gibt es zusätzliche alternative Adressierungsarten je nach Typ des verwendeten Stacks. Sie unterscheiden sich in der Adresse, auf die der Stapelzeiger weist [2]:

- Full Stacks** Der Stapelzeiger weist auf die zuletzt gefüllte (full) Adresse.
- Empty Stacks** Der Stapelzeiger weist auf die erste leere (empty) Adresse.
- Descending Stacks** Die Adresse des Stapelzeigers wird nach der Operation verringert.
- Ascending Stacks** Die Adresse des Stapelzeigers wird nach der Operation erhöht.

Diese Eigenschaften können kombiniert werden und man erhält einen Full Ascending (FA), Full Descending (FD), Empty Ascending (EA) oder Empty Descending (ED) Stack. Die alternativen Adressierungsarten werden dann zu einer der vier normalen Adressierungsarten umgewandelt. In welche Art sie umgewandelt werden, hängt auch davon ab, ob die Register gespeichert oder geladen werden. Beim Laden von einem FD Stack wird der Stapelzeiger nach der Operation erhöht (IA), aber beim Speichern auf einen FD Stack, muss der Stapelzeiger zuerst verringert werden, bevor das Register gespeichert werden kann (DB). Daraus ergeben sich dann folgende Adressierungsarten für die verschiedenen Stacks und Operationen [2]:

Stack-Adressierungsart	Standard-Adressierungsart	L–Bit	P–Bit	U–Bit
LDMFA	LDMDA	1	0	0
LDMFD	LDMIA	1	0	1
LDMEA	LDMDB	1	1	0
LDMED	LDMIB	1	1	1
STMED	STMDA	0	0	0
STMEA	STMIA	0	0	1
STMFD	STMDB	0	1	0
STMFA	STMIB	0	1	1

Tabelle 3: Umwandlung der Stack-Adressierungsarten in normale Adressierungsarten und gesetzte Bits in der Kodierung [2]



## 2.2 Parsing Expression Grammatik und tsPEG

Um die Benutzereingabe zu parsen habe ich tsPEG [5], einen Parser–Generator für TypeScript [10] verwendet. Dieser benutzt eine Parsing Expression Grammatik (PEG) [7], um dies so effizient wie möglich zu machen. Bei PEGs werden Mehrdeutigkeiten bei der Spezifikation der Grammatik vermieden. Dadurch kann für jede Grammatik ein Parser erstellt werden, mit dem beliebiger Text in linearer Zeit geparkt werden kann [7].

CFG	PEG
1. A --> a   a b	1. A <-- a / a b
2. A --> a b   a	2. A <-- a b / a

Listing 1: Definition einer Grammatik bei kontextfreien Grammatiken (links) und Parsing Expression Grammatiken (rechts) [7]

In kontextfreien Grammatiken gibt es den ungeordneten Alternativen–Operator (unordered choice operator) | mit dem man bei den Regeln der Grammatik ein Wahl zwischen mehreren Möglichkeiten beschreiben kann, wie in Listing 1 auf der linken Seite. Da der Operator ungeordnet ist, sind beide Definitionen gleichwertig. Dies lässt aber Mehrdeutigkeiten zu, da ein Parser alle alternativen Möglichkeiten betrachten muss, bevor er weiß, welche für einen konkreten Fall zutrifft. Damit kann man zwar sehr komplexe Grammatiken definieren, es führt aber auch zu nicht–linearen Zeiten um den Inhalt zu parsen [9].

Beim vielen Anwendungen ist jedoch ein schnelles Parsen von Bedeutung, besonders bei dem in dieser Arbeit beschriebenen Simulator, da Benutzer:innen eine schnelle Antwort erwarten, wenn sie ihren Code vom Simulator parsen lassen. Darum wird bei Parsing Expression Grammatiken der ungeordnete Alternativen–Operator durch den Alternativen–Operator mit Priorität / ersetzt. Bei diesem werden die Alternativen in absteigender Reihenfolge nach Priorität angegeben, wie in Listing 1 auf der rechten Seite. Bei einem Operator mit Priorität sind diese beiden Definitionen nicht gleichwertig. Bei der ersten Definition ist die zweite Alternative überflüssig, da der Parser, nachdem er *a* gefunden hat, die andere Alternative nicht mehr überprüft und diese somit nie erfüllt sein kann. Bei der zweiten Definition sucht der Parser zuerst nach *a b* und falls er dies nicht findet, weicht er auf die nächste Alternative mit niedriger Priorität *a* aus [7].

```
start := helloChoice

helloChoice := hello planet='Planet[0-9]' | helloWorld
helloWorld := hello planet='World'
hello := 'Hello '
```

Listing 2: Angepasstes Hello World Beispiel für tsPEG [5]

Die weiteren Definition und Operatoren werden in der Syntax von tsPEG erklärt, da die von mir geschriebene Grammatik in Appendix A auch diese Syntax aufweist. Listing 2 zeigt dafür ein einfaches Beispiel. Die einzelnen Regeln der Grammatik werden mit := definiert und sie können Strings mit ASCII–Charakteren oder weitere Regeln enthalten. Für Strings können auch reguläre Ausdrücke benutzt werden, wie in Listing 2 mit 'Planet[0–9]'. In den eckigen Klammern können einzelne Charaktere oder Bereiche (Charaktere getrennt mit –) spezifiziert werden, die identifiziert werden sollen. Mit = können die identifizierten Ausdrücke Variablen zugewiesen werden, die bei erfolgreichem Parsen in einem abstrakten Syntax Baum gespeichert werden. Bei fehlgeschlagenem Parsen, wird ein Array mit erwarteten Übereinstimmungen (Matches) ausgegeben. | ist der vorhin beschriebene Operator für

Alternativen mit Priorität. Die Alternativen werden mit absteigender Priorität aufgeführt. In unserem Beispiel bedeutet das, dass zuerst versucht wird *hello planet*=’Planet[0-9]’ zu identifizieren, bevor es zur nächsten Regel *helloWorld* übergeht [5].

Für die einzelnen Ausdrücke gibt es noch folgende Operatoren [5]:

- ? Der ?-Operator wird verwendet, um Teile von Regeln optional zu machen. Dafür wird einfach ein ? an das Ende eines Ausdrucks gehängt.
- + Erlaubt 1 oder mehrere Exemplare des Ausdrucks.
- \* Erlaubt 0 oder mehrere Exemplare des Ausdrucks.
- ! Dieser Operator wird für negativen Lookahead verwendet. Damit kann ein Ausdruck angegeben werden, der nicht erlaubt ist und das Parsen fehlschlägt, wenn der Ausdruck an dieser Stelle gefunden wird.

Meine Herangehensweise beim Schreiben der Grammatik für ARMv5 ist in Abschnitt 3.3 genauer beschrieben.

### 3 Implementation

Das Backend des Simulators wurde in TypeScript [10][3] geschrieben und als Frontend wurde das Webframework React [6] verwendet.

In diesem Abschnitt beschreibe ich zuerst die Klassen mit den Operanden und Instruktionen, meine Herangehensweise an die Implementation und dokumentiere die einzelnen Module des Programms. Nach einem kurzen Überblick gehe ich die wichtigsten Komponenten in der Reihenfolge durch, in der sie Benutzer:innen antreffen. Zuerst schreiben diese ihren Code in die Benutzereingabe, welche dann vom Parser ausgewertet und in den Hauptspeicher geschrieben wird. Die Instruktionen werden dann von der Code Execution Engine ausgeführt.

#### 3.1 Instruktionen und Operanden

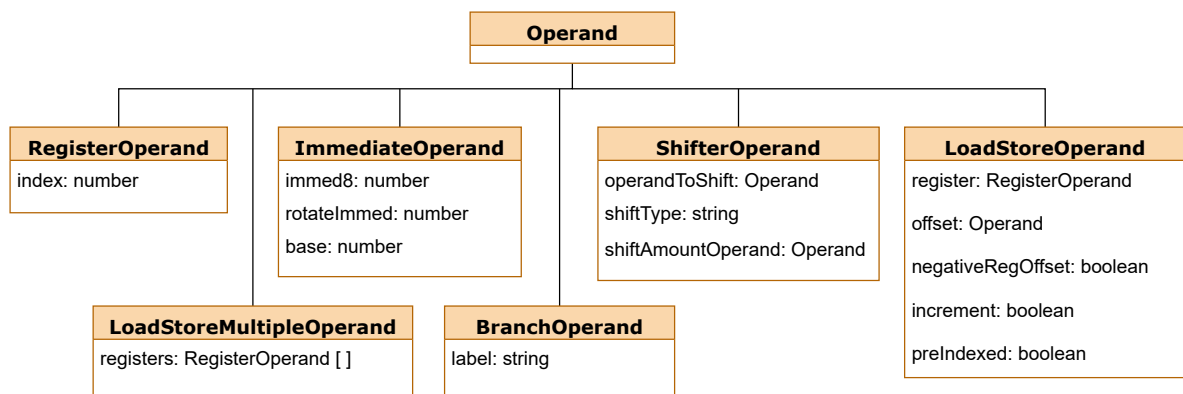


Abbildung 14: Alle Operanden–Klassen des Simulators und deren Felder.

Beginnen wir zuerst mit den kleinsten von mir definierten Datentypen, den Operanden. Diese werden dann verwendet um Instruktionen zu definieren. Abbildung 14 zeigt alle verschiedenen Klassen für Operanden und ihre Felder an. Ganz oben steht die Elternklasse *Operand*, von der alle Operanden ableiten. Diese Elternklasse dient dazu, Funktionen und Felder anderer Klassen mit einem allgemeinen Operanden zu definieren. Beim Aufrufen der Funktionen kann dann eine Unterscheidung zwischen den Subklassen getroffen werden und jeden Operanden separat behandeln. Dabei wird zwischen folgenden Subklassen unterschieden:

- *RegisterOperand*

Spezifiziert eines der 16 Register von ARM (r0–r15) und besitzt als einziges Feld den Index des Registers.

- *ImmediateOperand*

Spezifiziert die in Abschnitt 2.1.3.1 beschriebenen Immediate–Werte. Das Feld *immed8* gibt den 8–Bit Wert an und *rotateImmed* die Rotation von 0,2,4,6,...,28,30. Das *base* Feld dient lediglich zur schöneren Ausgabe in der richtigen Basis (Dezimal, Hex – 0x, Binär – 0b, Oktal – 0o).

- *ShifterOperand*

Ist der flexible 2. Operand von Instruktionen, der Zugriff auf den Barrel-Shiftter hat. *operandToShift* ist der Operand, der verschoben wird. *shiftType* ist eine der in Abbildung 6 gezeigten Verschiebungen/Rotationen und *shiftAmountOperand* der Operand mit der Anzahl an Bits, die verschoben/rotiert werden.

- *BranchOperand*

Operand für Sprunginstruktionen. Besitzt 1 Feld mit dem Label, zu dem gesprungen wird.

- *LoadStoreOperand* – Vergleiche mit Adressierungsarten in Abschnitt 2.1.6.1.

*register* beinhaltet das Basisregister und *offset* den Operanden für die Adressierungsart. Die restlichen Felder mit booleschen Werten entsprechen den Options-Bits (*preIndexed* = P-Bit für Unterscheidung zwischen pre- und post-indexed, *negativeRegOffset* = U-Bit für Richtung des Offsets, *increment* = W-Bit um die aktualisierte Adresse mit Offset zurück in das Basisregister zu schreiben).

- *LoadStoreMultipleOperand*

Beinhaltet als einziges Feld eine Liste mit allen für die Speicher-/Ladeoperation relevanten Registern. Die Registerliste wird im Konstruktor zusätzlich noch einmal sortiert und von Duplikaten befreit, um Fehler bei den Instruktionen zu vermeiden.

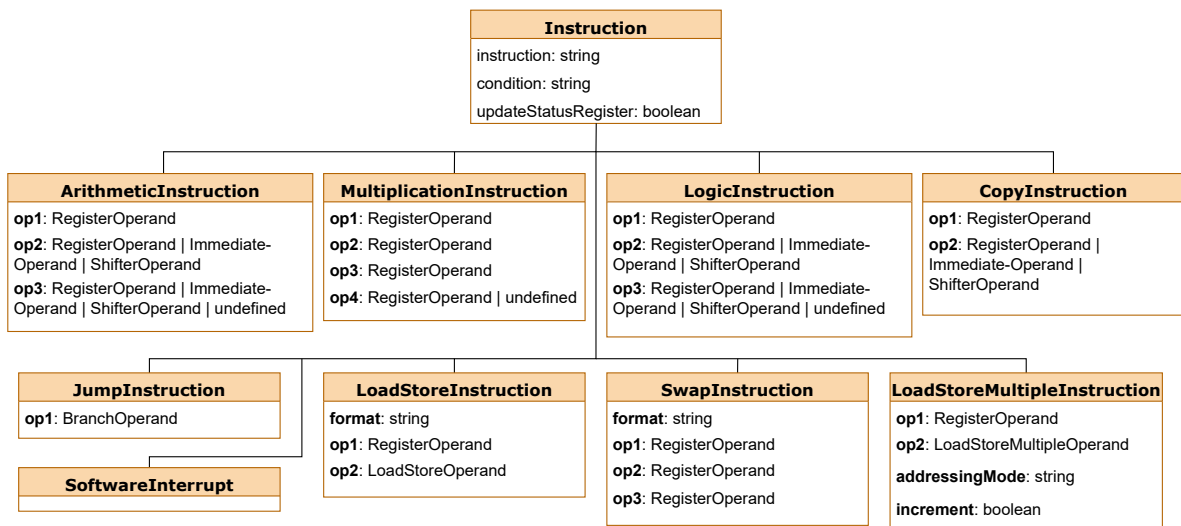


Abbildung 15: Alle Instruktions-Klassen des Simulators und deren Operanden.

Mit den Operanden können wir nun die Instruktionen von ARM definieren. Abbildung 15 zeigt die Elternklasse *Instruction* mit den allgemeinen Feldern, die alle Instruktionen gemeinsam haben und Subklassen, die nach Art der Instruktion aufgeteilt sind. Die Klassen sind aufgeteilt in:

- *Instruction*

Elternklasse, von der alle Subklassen ableiten. Sie besitzt ein *instruction* Feld mit dem Namen/Mnemonik der Instruktion, ein *cond* Feld für Bedingungen (siehe Abschnitt 2.1.2) und ein *updateStatusRegister* Feld für das S–Bit, um das Statusregister zu aktualisieren.

- *ArithmeticInstruction*

Gehört zu den datenverarbeitenden Instruktionen (Abschnitt 2.1.3) und ist für alle arithmetischen Instruktionen zuständig. Wenn alle 3 Operanden angegeben werden, ist der erste und zweite Operand ein Register und der dritte Operand ein Register, Immediate–Wert oder Shifter–Operand. Werden nur 2 Operanden angegeben und der dritte ist undefiniert (z.B. bei ADD r0, r1), wird *op1* für den ersten und zweiten Operanden hergenommen und *op2* für den dritten.

- *MultiplicationInstruction*

Gehört ebenfalls zu den datenverarbeitenden Instruktionen, nimmt aber nur Register als Operanden an. Die Multiplikation benötigt 3 Register und die Multiplikation mit Addition 4 Register.

- *LogicInstruction*

Ähnlich wie *ArithmeticInstruction*, besitzt aber auch Instruktionen die immer nur 2 Register annehmen und kein Zielregister haben (CMP, CMN, TST, TEQ). Können mit eigener Klasse gesondert von den arithmetischen Instruktionen behandelt werden.

- *CopyInstruction*

Hat das Zielregister in *op1* und einen flexiblen Operanden in *op2*, der ins Zielregister kopiert wird.

- *JumpInstruction*

Hat nur einen *BranchOperand*, der das Label für die Sprunginstruktion enthält.

- *LoadStoreInstruction*

Hat ein Zielregister in *op1* und einen *LoadStoreOperand* in *op2*. Zusätzlich noch einen *format* String um die verschiedene Datentypen bei der Adressierung zu unterscheiden.

- *SwapInstrucution*

Hat 3 Register als Operanden mit dem Zielregister in *op1*, das zu speichernde Register in *op2* und die Adresse, von der geladen wird, in *op3*. Der *format* String unterscheidet zwischen einer Wort– und Byte–Adressierung.

- *LoadStoreMultipleInstruction*

Hat das Basisregister in *op1* und einen *LoadStoreMultipleOperand* in *op2*. Mit *addressingMode*

wird zwischen den Adressierungsarten aus Abschnitt 2.1.7.1 unterschieden und *increment* gibt an, ob die aktualisierte Adresse zurück in das Basisregister geschrieben wird.

- *SoftwareInterrupt*

Zuletzt gibt es noch eine Klasse für Software–Interrupts, die keine weiteren Felder hat. Sie ruft bei Ausführung eine Funktion auf, die den korrekten Interrupt je nach Inhalt der Register ausführt.

## 3.2 Übersicht

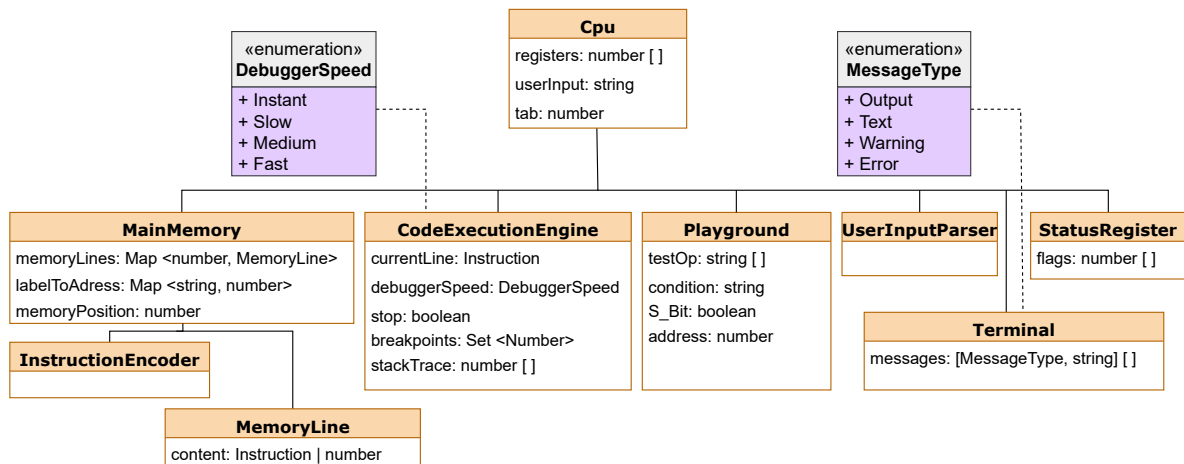


Abbildung 16: Klassendiagramm des Simulators und die wichtigsten Felder jeder Klasse.

In diesem Abschnitt gebe ich einen kurzen Überblick über die restlichen Klassen des Simulators und deren Felder, bevor ich auf die wichtigen Komponenten nochmal genauer eingehe. In Abbildung 16 ist ein Klassendiagramm dieser Klassen zu sehen.

- *Cpu*

Die Klasse für den Prozessor ist die Hauptkomponente des Simulators und beinhaltet alle weiteren Komponenten. In dieser Klasse ist auch der größte Teil der Benutzeroberfläche definiert. Die wichtigsten Felder dieser Klasse sind das *registers* Feld, das die 16 ARM Register enthält, das *userInput* Feld für die Benutzereingabe und das *tab* Feld für den derzeit geöffneten Tab (Feld zur Code–Eingabe oder Hauptspeicher).

- *MainMemory*

Der Hauptspeicher enthält im Feld *memoryLines* den Inhalt des Speichers in einer Map mit der Adresse als Schlüssel und einer *MemoryLine* als Wert. Das *labelToAddress* Feld ist ebenfalls eine Map mit dem Label als Schlüssel und der dazugehörigen Adresse als Wert. Das *memoryPosition* Feld enthält die derzeitige Adresse für die Ausführung und zum Hervorheben in der Benutzeroberfläche.

- *MemoryLine*

Definiert eine Zeile im Hauptspeicher, die entweder eine kodierte Instruktion oder einen beliebigen 32–Bit Wert als Daten enthält.

- *InstructionEncoder*

Wandelt die Instruktionen des Hauptspeichers in deren Kodierung um, um diese auf der Benutzeroberfläche darzustellen.

- *CodeExecutionEngine*

Klasse, um die Instruktionen im Hauptspeicher auszuführen. Enthält ein *currentLine* Feld mit der auszuführenden Instruktion, einem *debuggerSpeed* Feld für die Geschwindigkeit des Debuggers mit dazugehöriger Enumeration (Instant, Slow, Medium, Fast), einem *stop* Feld um die Ausführung anzuhalten, einem *breakPoint* Feld mit einem Set von Adressen, bei denen die Ausführung angehalten werden soll und einem *stackTrace* Feld, das ein Array mit den Adressen des Stacktrace enthält. Die Funktionsweise dieser Klasse wird in Abschnitt 3.5 besprochen.

- *Playground*

Ermöglicht den Benutzer:innen einzelne Instruktion auszuführen und dem Hauptspeicher hinzuzufügen. Enthält die dafür nötigen Felder mit Operanden, Bedingung, dem S-Bit und der Adresse an der die Instruktion hinzugefügt werden soll. Knöpfe für jede Instruktion lesen diese Felder, überprüfen sie auf Korrektheit und fügen die Instruktion hinzu.

- *UserInputParser*

Arbeitet den abstrakten Syntaxbaum nach Parsen der Benutzereingabe ab. Siehe Abschnitt 3.3.

- *StatusRegister*

Enthält die 4 Status-Flags für NZCV und Funktionen um diese basierend auf dem Ergebnis einer Instruktion zu aktualisieren.

- *Terminal*

Enthält im Feld *messages* ein Array mit Nachrichten und dazugehörigen Nachrichtentyp (Enumeration *MessageType* – Output, Text, Warning, Error), die auf dem Terminal ausgegeben werden.

### 3.3 Parser

Für das Parsen der Benutzereingabe habe ich eine Parsing Expression Grammatik [7] geschrieben. Die gesamte Grammatik ist im Appendix A zu finden. Im den folgenden Listings befinden sich vereinfachte Versionen der Grammatik, um den Aufbau leichter beschreiben zu können.

```
start := start=line  
  
line := label? directive comment? nextLine |  
        label? instruction comment? nextLine |  
        label? commentLine? nextLine |
```

## Listing 3: Einteilung in Zeilen

Meine Herangehensweise bei der Grammatik war das Einteilen des gesamten Codes in einzelne Zeilen, wie in Listing 3 dargestellt. Eine Zeile kann dabei immer ein optionales Label und ein Kommentar haben. Sie hat immer eine nächste Zeile, außer es ist die letzte Zeile des zu parsenden Codes. Die verschiedenen Arten einer Zeile werden weiter unterteilt. In Listing 3 gibt es Direktiven, Instruktionen und Kommentarzeilen.

```
directive := directive=ascii |
           directive=space |
           '.arm' | '.text' | '.data' |
           '.align' | '.global _start'

ascii := '.ascii' '"' data='[ -!#-~]*' '"'
space := '.space' size='[0-9]+'

```

## Listing 4: Direktiven

Listing 4 zeigt die Regeln für Direktiven. Für die *.ascii*-Direktive in der ersten Zeile wird eine weitere Regel definiert und die ASCII-Charaktere nach *'ascii'* werden in der Variable *data* gespeichert, um diese später beim Abarbeiten des Syntaxbaums in den Hauptspeicher schreiben zu können. Dasselbe gilt für die *.size*-Direktive, bei der die Größe des zu reservierenden Speichers in die *size* Variable eingelesen wird, um später diese Anzahl an Bytes im Hauptspeicher zu reservieren. Die übrigen Direktiven wie *'arm'* oder *'text'* benötigen keine weiteren Subregeln, da bei diesen nur der Text erkannt werden muss.

```
instruction := art | log | copyJump | loadStore | loadStoreMultiple |
              softwareInterrupt

art := inst=artInst cond=condition operands=artOp |
      inst='mul' cond=condition operands=artMulOp |
      inst='mla' cond=condition operands=artMlaOp

artInst := 'add' | 'adc' | 'sub' | 'sbc' | 'rsb' | 'rsc'

artOp := artOp3 | artOp2
artOp2 := op1=regOp ',' op2=op
artOp3 := op1=regOp ',' op2=regOp ',' op3=op

artMulOp := op1=regOp ',' op2=regOp ',' op3=regOp
artMlaOp := op1=regOp ',' op2=regOp ',' op3=regOp ',' op4=regOp

```

## Listing 5: Instruktionen

Bei Instruktionen findet eine Unterteilung in die Art der Instruktion statt. Jede Instruktion hat dabei den Namen der Operation, eine optionale Bedingung und Operanden. Je nach Operation gibt es eine unterschiedliche Anzahl an Operanden. Listing 5 zeigt die arithmetischen und multiplikativen Operationen. Die *artOp*-Regel verlangt entweder 2 oder 3 Operanden (Vergleiche Abschnitt 3.1 → bei 2 Operanden wird der erste Operand dupliziert und für Operand 1 und 2 hergenommen). Der erste Operand ist dabei immer ein Register und der letzte kann ein flexibler Operand (Register, Immediate-Wert oder Shifter-Operand) sein. Die Regeln für Multiplikation *artMulOp* und *artMlaOp* lassen nur Register zu, da diese Instruktionen nur mit Registern arbeiten können.



```

op := shiftOp | regImmOp
regImmOp := regOp | immOp
shiftOp := opToShift=regImmOp ',' shiftType=shiftType opShift=regImmOp

regOp := regOp='[rR][0-9]+' | 'pc' | 'lr' | 'sp'

immOp := immType sign base number='[0-9a-fA-F]+'
immType := '#' | '='
base := '0x' | '0b' | '0o' | ''
sign := '-' | '+' | ''

```

Listing 6: Operanden

Zuletzt müssen dann natürlich die Regeln für die Operanden aus Abschnitt 3.1 definiert werden. Listing 6 zeigt die Regeln für Register, Immediate–Werte und Shifter–Operanden. Bei der Regel für *regOp* wird ein 'r' gefolgt von einer beliebigen Zahl, 'pc', 'lr' oder 'sp' erwartet. Bei der Regel für Immediate–Werte *immOp* wird zusätzlich zu der Zahl noch der Typ, eine Basis und ein Vorzeichen eingelesen. Die obersten drei Regeln zeigen wie diese Basis–Operanden kombiniert werden können. *op* lässt alle 3 Operanden zu. Die Regel *regImmOp* existiert, da bei der Regel für Shifter–Operanden nur Register oder Immediate–Werte vorkommen dürfen. Die Regeln für die restlichen Operanden und etwaige Hilfsregeln sind in der Grammatik im Appendix A zu finden.

```

1 let line = ast.start;
2
3 while (line.kind !== ASTKinds.line_5) {
4   let currentLine = line.currentLine;
5
6   switch (currentLine.kind) {
7     case ASTKinds.instruction_1: this.parseArithmeticInstruction(currentLine.instruction); break;
8     case ASTKinds.instruction_2: this.parseLogicInstruction(currentLine.instruction); break;
9     ...
10    case ASTKinds.directive_1: this.addASCIIData(currentLine.directive.data); break;
11    case ASTKinds.directive_2: this.addData(currentLine.directive.size, "0"); break;
12    ...
13  }
14
15  line = line.nextLine;
16 }

```

Listing 7: Abarbeiten des abstrakten Syntaxbaumes in UserInputParser.ts

Aus dieser Grammatik generiert das in Abschnitt 2.2 beschriebene tsPEG [5] einen Parser für TypeScript. Der generierte Parser kann nun die Korrektheit der Benutzereingabe überprüfen und erstellt bei korrektem Code einen abstrakten Syntaxbaum aus den zugewiesenen Elementen in der Grammatik. Dieser kann dann Zeile für Zeile wie in Listing 7 abgearbeitet werden. Zuerst wird der jetzigen Zeile der Start des Syntaxbaumes zugewiesen. In einer while–Schleife wird dann die passende Funktion für die Art der Zeile aufgerufen und in die nächste Zeile gewechselt. Dies geschieht solange, bis die letzte Zeile erreicht ist.

### 3.4 Hauptspeicher

Wie in Abschnitt 3.2 beschrieben, sind die Zeilen des Hauptspeichers eine Map mit der Adresse als Schlüssel und einer *MemoryLine* (enthält eine Instruktion oder Daten in Form einer 32-Bit Zahl) als Wert. Bei Adressen, die nicht in der Map sind, wird eine neue *MemoryLine* mit dem Standardwert 0x00000000 (oder ein beliebigen anderen Wert) zurückgegeben. Um eine Instruktion in den Hauptspeicher zu schreiben, gibt es die in Listing 8 beschriebene Funktion (vereinfachtes Beispiel im Fall einer arithmetischen Instruktion). Dieser Funktion werden alle nötigen Parameter übergeben, um jede Instruktion erstellen zu können.

```
1 addInstruction(instruction: string, condition: string, updateStatusRegister: boolean,
2   op1String: string | undefined, op2String: string | undefined, op3String: string | undefined,
3   op4String: string | undefined, address?: number): boolean {
4
5   let newInstruction;
6
7   if (["add", "adc", "sub", "sbc", "rsb", "rsc"].includes(instruction)) {
8     let op1 = this.addRegisterOperand(op1String);
9     let op2 = this.addRegImmShiftOperand(op2String);
10    let op3 = this.addRegImmShiftOperand(op3String);
11
12    if (op1 !== undefined && op2 !== undefined && op3 !== undefined) {
13      newInstruction =
14        new ArithmeticInstruction(instruction, condition, op1, op2, op3, updateStatusRegister);
15    }
16  }
17  ...
18  if (typeof newInstruction !== 'undefined') {
19    this.memoryLines.set((this.memoryLines.size * 4), new MemoryLine(newInstruction));
20    return true;
21  }
22 }
```

Listing 8: Hinzufügen einer Instruktion am Beispiel einer arithmetischen Instruktion

In Zeile 5 wird zuerst eine undefinierte Variable *newInstruction* für eine neue Instruktion erstellt. Dann wird in Zeile 7 überprüft, um welche Art von Instruktion es sich handelt. In Zeilen 8–10 werden dann aus den übergebenen Strings für die Operanden die Objekte für die Operanden aus Abschnitt 3.1 erstellt. War dies für alle nötigen Operanden erfolgreich (Zeile 12), wird der Variablen *newInstruction* eine arithmetische Instruktion mit den übergebenen Parametern zugewiesen. Am Ende der Funktion (Zeile 18) wird dann überprüft, ob die Variable definiert ist und dann zu der Map mit den Zeilen des Hauptspeichers hinzugefügt. Die Instruktion wird beim Kompilieren der ersten leeren Adresszeile (4 \* Anzahl der Adresszeilen) zugewiesen. Wird der letzte Parameter *address* der Funktion angegeben (z.B. in der Playground-Komponente), kann die Instruktion an eine beliebige Stelle gespeichert werden. Die Funktion gibt dann einen booleschen Wert zurück, ob das Hinzufügen der Instruktion erfolgreich war.

Ähnliche Funktionen gibt es auch für das Speichern von beliebigen Daten in den Hauptspeicher oder das Hinzufügen von Labeln, die einer bestimmten Adresse zugeordnet sind. Ist der Hauptspeicher dann mit Instruktionen und Daten gefüllt, wird dieser wie in Abbildung 17 gerendert und in der Benutzeroberfläche dargestellt. In der ersten Spalte können Benutzer:innen Breakpoints für die jeweilige Zeile setzen. In der zweiten Spalte wird die Adresse der Zeile angeführt. In der dritten Zeile wird entweder die Kodierung der Instruktion oder die dort gespeicherten Daten dargestellt. Bei Instruktionen wird zusätzlich in einer vierten Spalte die Instruktion zu der Kodierung angeführt. Falls eine Zeile ein Label besitzt, wird dies

oberhalb der Zeile angezeigt.

Bei dem Beispiel in Abbildung 17 handelt es sich um das Hello World Beispiel für ARM. In diesem Fall sind in den Zeilen nach dem Label 'msg' ASCII-Daten gespeichert ("Hello Innsbruck!\n") und die Instruktionen nach dem Label '\_start' laden diese Daten in die Register und geben sie mittels Software-Interrupt aus. Damit können Benutzer auch erkennen, ob es sich lediglich um Daten oder eine Instruktion handelt, da Instruktionen immer neben der Zeile mit ihrer Kodierung stehen.

●	Address	Encoding	Instruction
		<b>msg:</b>	
	00000000	6c6c6548	
	00000004	6e49206f	
	00000008	7262736e	
	0000000c	216b6375	
	00000010	0000000a	
		<b>_start:</b>	
	00000014	e3a00001	mov r0, #1
	00000018	e0000000	ldr r1, =msg
●	0000001c	e0000000	ldr r2, =len
	00000020	e3a07004	mov r7, #4
●	00000024	ef000000	swi #0
	00000028	e3a00000	mov r0, #0
	0000002c	e3a07001	mov r7, #1
	00000030	ef000000	swi #0

Abbildung 17: Ausschnitt aus dem Hauptspeicher des Simulators

### 3.5 Code Execution Engine

Die nächste große Komponente ist die Code Execution Engine, die auch als Debugger für den Simulator fungiert. Sie ist für das Ausführen der Instruktionen, die zuvor in den Hauptspeicher geschrieben wurden, zuständig. Sie beinhaltet die in Abschnitt 3.2 beschriebenen Felder (aktuelle Instruktion, Variable für die Debugger Geschwindigkeit, einen booleschen Wert zum Stoppen der Ausführung, ein Set mit Breakpoints und den Stacktrace), sowie zusätzliche Felder um temporär die Register, das Statusregister und den Inhalt des Hauptspeichers in dieser Komponente zu speichern. Diese Felder zum temporären Speichern sind wichtig, da sonst nach jeder Instruktion die Felder in den jeweiligen Klassen aktualisiert werden, was einen großen Overhead verursacht. Wenn die Geschwindigkeit des Debuggers auf *Instant* gesetzt ist, werden alle Instruktionen (bis zu eventuellen Breakpoints) beim Klicken des Continue-Buttons sofort ausgeführt und erst am Ende werden die Felder in den dazugehörigen Klassen aktualisiert. Beim Drücken eines Debugger-Buttons zum Ausführen einer oder mehrerer Instruktion wird immer zuerst die *continue()*-Funktion aufgerufen. Diese ruft dann die *execNextInstruction()*-Funktion auf, bei der die aktuelle Adresse überprüft und die aktuelle Instruktion aus dem Hauptspeicher geladen wird. Bei korrekt ausgerichteter Adresse wird dann von dieser Funktion *executeInstruction()* aufgerufen, welche dann schließlich die eigentliche Instruktion ausführt.

```

1 continue = async () => {
2   this.newRegisters = [...this.cpu.state.registers];
3   this.newStatusRegister = this.cpu.state.statusRegister;
4   this.newMainMemory = this.cpu.state.mainMemory;
5
6   let endOfSubroutine = false;
7   let stackSizeEndSubroutine = this.stackTrace.length - 1;
8
9   let currentNumInstruction = 0;
10  do {
11    if (currentNumInstruction++ > this.maxContinueInstructions) {
12      break;
13    }
14    // wait if speed not Instant
15    if (this.debuggerSpeed !== DebuggerSpeed.Instant) {
16      await this.delay(this.debuggerSpeed.valueOf())
17      this.cpu.setState({ registers: this.newRegisters, statusRegister: this.newStatusRegister,
18        mainMemory: this.newMainMemory });
19    }
20    // return from subroutine
21    if (this.stopSubroutine && (this.stackTrace.length === stackSizeEndSubroutine)) {
22      endOfSubroutine = true;
23      this.stopSubroutine = false;
24    }
25  } while (!endOfSubroutine && this.executeNextInstruction() && !this.stop);
26
27  // update register at the end, if speed is Instant
28  this.cpu.setState({ registers: this.newRegisters, statusRegister: this.newStatusRegister,
29    mainMemory: this.newMainMemory });
30 }

```

Listing 9: Continue–Funktion der Code Execution Engine

Diese *continue()*–Funktion ist in Listing 9 zu sehen. Es handelt sich dabei um eine asynchrone Funktion, da bei den Debugger-Geschwindigkeiten *Slow*, *Medium* und *Fast* eine bestimmte Zeit gewartet wird, bis die nächste Funktion ausgeführt wird. In Zeilen 2–4 werden die Register, das Statusregister und der Hauptspeicher in den zuvor beschriebenen Feldern zur temporären Speicherung hinterlegt. In Zeilen 6–7 werden zwei Variablen für das Ausführen bis zum Ende einer Subroutine deklariert. Das Ende einer Subroutine ausgehend von der derzeitigen Instruktion ist erreicht, wenn sich die Größe des aktuellen Stacktraces um 1 verringert hat. Zeile 9 initialisiert einen Zähler für ausgeführte Instruktionen, um nach Ausführung einer festgelegten Anzahl an Instruktionen zu unterbrechen. Falls Benutzer:innen Code schreiben, der in einer Endlosschleife endet, unterbricht dies die Ausführung und warnt Benutzer:innen vor einer möglichen Schleife. Dies wird in der ersten If–Abfrage in Zeile 11 innerhalb der Schleife gemacht. Die Warnungen und Fehlerbenachrichtigungen wurden in den Code–Ausschnitten der Arbeit entfernt, um diese kürzer zu halten. Die nächste If–Abfrage in Zeile 15 überprüft die Debugger-Geschwindigkeit. Ist diese nicht *Instant*, wird die vorgegebene Zeit gewartet und die Register, das Statusregister und der Hauptspeicher aktualisiert. Die letzte If–Abfrage in Zeile 21 wird ausgeführt falls das Feld für *stopSubroutine* auf *true* gesetzt ist und überprüft, ob das Ende der Subroutine erreicht ist. In der While–Schleife in Zeile 25 wird dann zuerst überprüft, ob das Ende der Subroutine erreicht ist. Ist dies nicht erreicht, wird die nächste Instruktion ausgeführt und danach überprüft ob nach dieser Instruktion gestoppt werden soll (z.B. durch Breakpoints). In den letzten Zeilen 27–28 wird nochmal der Zustand aller Komponenten aktualisiert.

```

1 executeNextInstruction(): boolean {
2     let memoryAddress = this.newRegisters[15];
3
4     // check for aligned memory address
5     if (memoryAddress % 4 === 0 && typeof this.newMainMemory !== 'undefined') {
6         // execute instruction
7         this.currentLine = this.newMainMemory.getMemoryLine(memoryAddress).getContent();
8         let successful = this.executeInstruction();
9         // set stop, if there is a breakpoint
10        if (this.breakpoints.has(this.newRegisters[15])) {
11            this.stop = true;
12        }
13
14        // update last element of current stack trace
15        this.stackTrace[this.stackTrace.length - 1] = this.newRegisters[15]
16
17        return successful;
18    }
19    // unaligned address
20    else {
21        this.cpu.newTerminalMessage("Invalid Memory Address!", MessageType.Error);
22        return false;
23    }
24 }

```

Listing 10: Funktion zum Ausführen der nächsten Instruktion

```

1 executeInstruction(): boolean {
2     // get currentline and increase PC
3     let inst = this.currentLine;
4     this.newRegisters[15] += 4;
5
6     let condition = inst.getCondition();
7     let flags = this.newStatusRegister.getFlags();
8
9     switch (condition) {
10        case "eq": if (flags[1]) { break; } return true;
11        case "ne": if (!flags[1]) { break; } return true;
12        ...
13        case "al": break;
14        case "nv": return true;
15    }
16
17    if (inst instanceof ArithmeticInstruction) {
18        result = this.arithmetic(inst, op2, op3, op4);
19    }
20    ...
21    else if (inst instanceof SoftwareInterrupt) {
22        this.softwareInterrupt();
23    }
24 }

```

Listing 11: Funktion zum Ausführen der aktuellen Instruktion je nach Art der Instruktion

Die *execNextInstruction()*–Funktion um die nächste Instruktion Auszuführen ist in Listing 10 zu finden. Diese Funktion gibt immer einen booleschen Wert zurück, ob die Ausführung der Instruktion erfolgreich war, der dann in der While-Schleife der *Continue()*–Funktion überprüft werden kann. Zuerst holt sich die Funktion den aktuellen Wert des Befehlszählers

in Zeile 2. Dann wird überprüft ob diese Adresse korrekt ausgerichtet ist und ansonsten eine Fehlermeldung auf dem Terminal ausgegeben (Zeile 21). In Zeile 7 wird der aktuellen Zeile die Instruktion aus dem Hauptspeicher zugewiesen und dann in Zeile 8 ausgeführt. Danach wird auf einen Breakpoint überprüft und der Stacktrace aktualisiert.

Die Funktion zum Ausführen der aktuellen Instruktion *executeInstruction()* ist in Listing 11 dargestellt. Zuerst wird in Zeile 4 der Befehlszähler erhöht. In Zeilen 10-14 werden die in Abschnitt 2.1.2 eingeführten Bedingungen mit den aktuelle Status-Flags verglichen. Diese bestimmen dann ob die die Instruktion überhaupt ausgeführt werden soll. Danach werden in Zeilen 17-23, je nach Art der Instruktion, die jeweiligen Funktionen aufgerufen, welche dann die in Abschnitt 2.1 beschriebenen Operationen ausführen.

### 3.6 Benutzeroberfläche

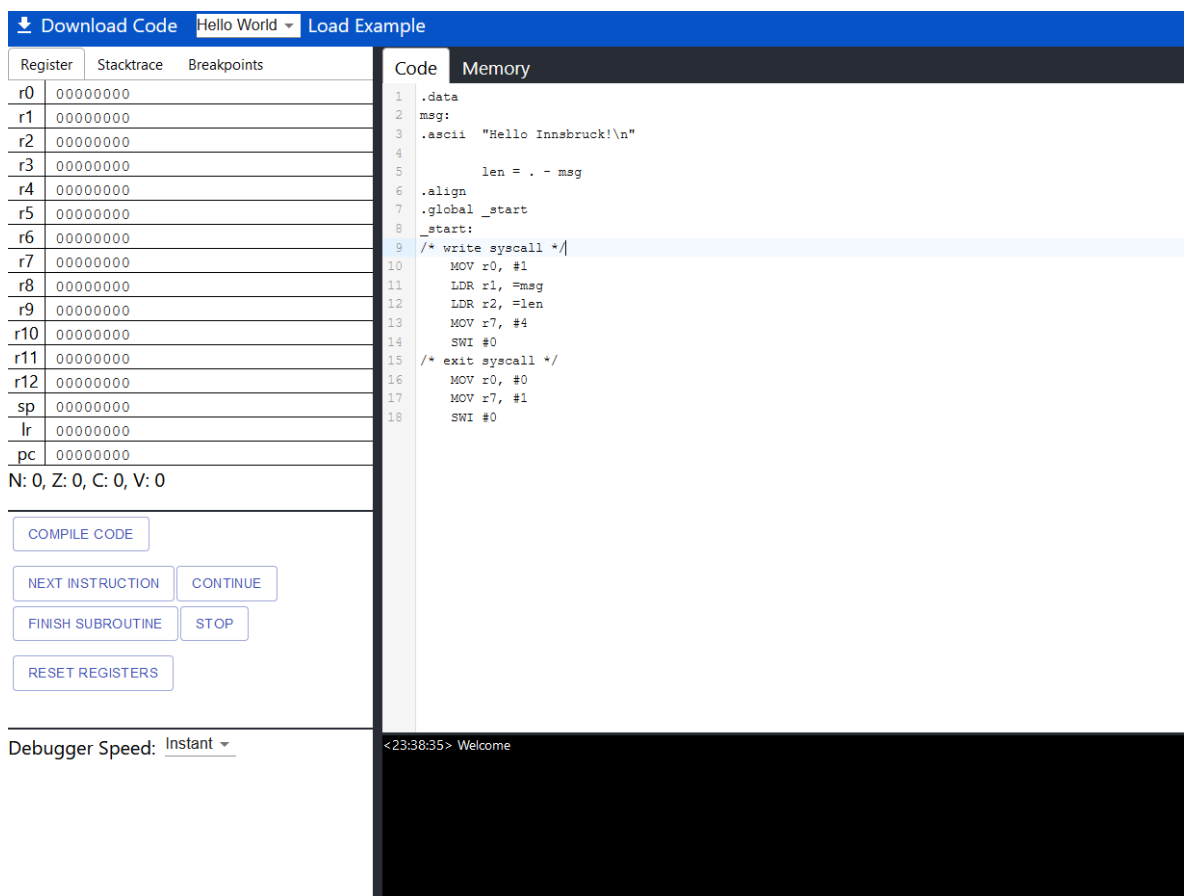


Abbildung 18: Hauptansicht der Benutzeroberfläche

Die Startseite des ARM Simulators ist in Abbildung 18 zu sehen. Das Layout ist von ähnlichen Simulatoren, wie z.B. CPUlator [14] inspiriert.

Oben im Header gibt es einen Button mit dem Benutzer:innen den aktuelle Code in der Benutzereingabe herunterladen können. Daneben befindet sich ein Dropdown-Menü mit dem man verschiedene Beispiele laden kann. Der oberste Bereich im linken Abschnitt zeigt den aktuellen Inhalt der Register und die Status-Flags. Der Inhalt der einzelnen Register kann von Benutzer:innen verändert werden. Darunter befindet sich der Bereich für den Debug-

ger und Parser. Mit dem 'Compile Code'–Button kann die aktuelle Benutzereingabe geparkt werden. Nach erfolgreichem Parsen, werden die Instruktionen und Daten in den Hauptspeicher geschrieben und der 'Code'–Tab wechselt auf den 'Memory'–Tab. Die nächsten 4 Knöpfe sind zum Starten bzw. Stoppen der Ausführung. Mit 'Next Instruction' wird nur die nächste Instruktion ausgeführt und mit 'Continue' werden solange Instruktionen ausgeführt, bis der Debugger auf einen Breakpoint oder eine invalide Instruktion trifft. 'Finish Subroutine' funktioniert ähnlich wie 'Continue', stoppt aber zusätzlich nach Beendigung einer Subroutine. Mit 'Stop' kann die Ausführung an der aktuellen Stelle gestoppt werden. Im Gegensatz zu den im Exposé erwähnten Namen für diese Buttons ('Step Into', 'Step Over'), habe ich mich für aussagekräftigere Namen entschieden, um den Einstieg in den Debugger zu erleichtern. Mit 'Reset Register' werden alle Register zurück auf 0 gesetzt. Darunter befindet sich noch der Bereich für Optionen, bei man aktuell nur die Geschwindigkeit des Debuggers einstellen kann. Diese Option beeinflusst wie schnell 'Continue' und 'Finish Subroutine' die Instruktionen abarbeiten. Auf der rechten Seite gibt es ein großes Feld für die Benutzereingabe und darunter ein Terminal für Output von Software–Interrupts und Warnungen/Fehlermeldungen des Simulators.

Register	Stacktrace	Breakpoints
	00000008	
	00000068	
	00000068	
	00000068	
	0000004c	

Register	Stacktrace	Breakpoints
	00000014	✕
	00000034	✕
	0000004c	✕
	00000058	✕
	0000005c	✕

Abbildung 19: Stacktrace– und Breakpoints–Tab

Im linken oberen Bereich können Benutzer:innen auf andere Tabs umschalten (Abbildung 19). Der zweite Tab beinhaltet den Stacktrace. Dort wird bei Sprüngen mit Hinterlegung der Rücksprungadresse (BL) die Adresse der Sprunginstruktion hinterlegt und beim Kopieren der Adresse im Link–Registers in das Register für den Befehlszähler (MOV pc, lr) wird der unterste Eintrag des Stack entfernt (pop). Der unterste Eintrag enthält also immer die aktuelle Adresse im Befehlszähler und der darüber liegende Eintrag die Adresse vor dem Sprung in die Subroutine. Der dritte Tab zeigt alle Breakpoints an, die im 'Memory'–Tab gesetzt wurden in sortierter Reihenfolge an. Die einzelnen Breakpoints können an dieser Stelle mit dem Button rechts von den Adresses der Breakpoints entfernt werden.

Zuletzt gibt es noch den Memory–Tab mit dazugehörigem Playground (Abbildung 20). Der Inhalt der Adresszeilen wurde bereits in Abschnitt 3.4 erklärt. Die zusätzlichen Funktionen befinden sich alle im Header dieses Tabs. Zuerst gibt es einen 'GOTO'–Button mit dazugehörigem Adressfeld. Wird auf den Button geklickt, springt die im Adressfeld spezifizierte Adresse in den Fokus. Rechts daneben gibt es noch separate Knöpfe um zur Adresse des Stapelzeigers, des Link–Registers oder des Befehlszählers zu springen. Weiter rechts gibt es den 'CLEAR MEMORY'–Button um den Inhalt des Hauptspeichers zurückzusetzen. Daneben ist noch der 'PLAYGROUND'–Button, bei dem die in Abschnitt 3.2 beschriebene *Playground*–Komponente aufpoppt. Benutzer:innen können dort alle Operanden und Optionen von Instruktionen angeben und dann mit den Buttons für die einzelnen Instruktionen

werden diese in den Hauptspeicher an die im Adressfeld spezifizierte Adresse geschrieben und optional gleich ausführt.

The screenshot displays a memory editor interface with a table of memory contents and a playground for adding instructions.

Address	Encoding	Instruction
00000000	Start:	
00000000	e0000000	ldr r0, =1024
00000004	e0000000	ldr r1, =245
00000008	e3a02000	mov r2, #0
0000000c	eb000006	bl div32
00000010	e1a04000	mov r4, r0
00000014	eb00000e	bl hex
00000018	e1a04002	mov r4, r2
0000001c	eb00000c	bl hex
00000020	e3a00000	mov r0, #0
00000024	e3a07001	mov r7, #1
00000028	ef000000	swi #0
0000002c	div32:	
0000002c	e3a03020	mov r3, #32
00000030	div32loop:	
00000030	e1b00080	movs r0, r0, lsl #1
00000034	e1a02082	mov r2, r2, lsl #1
00000038	e2a22000	adc r2, r2, #0
0000003c	e0712002	rabs r2, r1, r2
00000040	53800001	orrpl r0, r0, #1
00000044	40822001	addmi r2, r2, r1
00000048	e2533001	subs r3, r3, #1
0000004c	1a0ffff7	bne div32loop
00000050	e1a0f00e	mov pc, lr
00000054	hex:	
00000054	e92d5fff	stmfd sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}
00000058	e3a03008	mov r3, #8
0000005c	e3a07004	mov r7, #4
00000060	e3a00001	mov r0, #1
00000064	e3a02001	mov r2, #1
00000068	hexloop:	
00000068	e0000000	ldr r1, =lut

**Playground Panel:**

**Add Instructions to Memory**  
Uses maximum number of possible operands for instruction (e.g. op3 has to be empty for ADD r0, r1)

Op1:   
 Op2:   
 Op3:   
 Op4:   
 Cond:   
 S: ☐  
 Address(Hex):   
 Execute Instruction: ☐

Buttons: ADD, ADC, SUB, SBC, RSB, RSC, MUL, MLA, AND, ORR, EOR, BIC, CMP, CMN, TST, TEQ, MOV, MVN, B, BL, ADD LABEL (OP1 AT ADDRESS), LDR, STR, RESET MEMORY

Abbildung 20: Memory-Tab und Playground



## 4 Evaluation

Die Korrektheit des Simulators wurde an den Beispielen aus der Vorlesung und dem Proseminar zur Einführung in die Technische Informatik aus dem Wintersemester 2018/19 getestet. Die konkreten Beispiele können im Simulator mit dem Dropdown-Menü 'Load Example' geladen und ausprobiert werden. Alle Beispiele liefern die gleichen Ergebnisse, wie die Kompilierung und Ausführung mit QEMU [13] auf dem Windows Subsystem for Linux [11]. Für umfangreiches Testen mit Unittests war im Rahmen dieser Bachelorarbeit leider keine Zeit mehr.

Um die Performance zu testen, habe ich die Ausführungszeit von zwei Programmen für eine große Anzahl an Instruktionen gemessen. Die Benchmarks wurden auf meinem Desktop-PC unter Windows 10 Enterprise – 64 Bit mit einer 'Intel Core i5-4450 CPU @ 3.20GHz' und 16GB RAM durchgeführt.

Das erste Programm ist das Divisions-Programm in Appendix B.1. Dieses Programm wurde gewählt um die Ausführung von Instruktionen, die nur eine Operation ausführen (kein Laden/Speichern von mehreren Registern) und wenig mit dem Stack arbeiten, zu testen. Um eine große Anzahl an Instruktionen zu erzielen, wird am Ende des Programms einfach wieder an den Anfang gesprungen und das Programm erneut ausgeführt, bis die geplante Anzahl an Instruktionen erreicht wurde. Das Ergebnis des Benchmarks für dieses Programm ist in Abbildung 21 illustriert. Daraus lassen sich eine durchschnittliche Ausführung von 507000 Instruktionen/s für Firefox und 616000 Instruktionen/s für Chrome berechnen. Mit Chrome lassen sich also 21.5% mehr solcher Instruktionen pro Sekunde ausführen als mit Firefox.

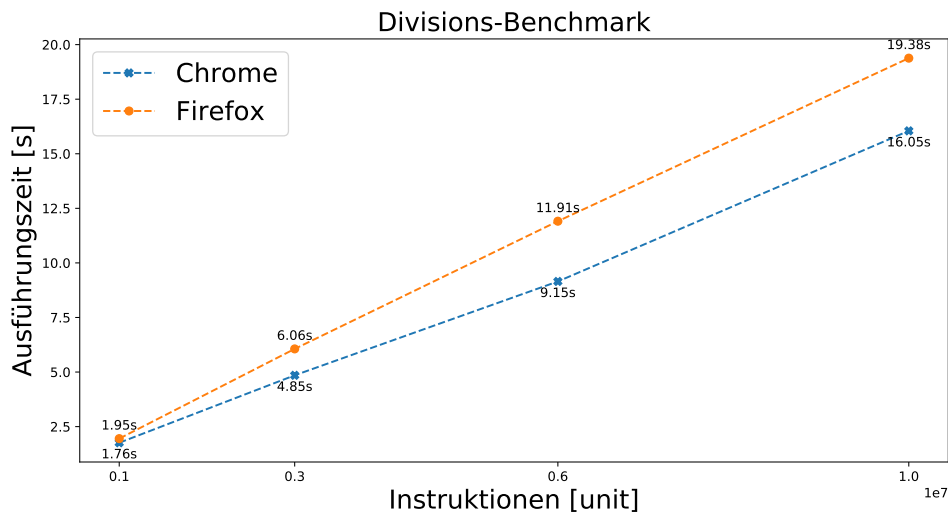


Abbildung 21: Benchmark-Test für das Divisions-Programm in Appendix B.1. Getestet wurde die Zeit (Durchschnitt von 5 Durchläufen) für eine bestimmte Anzahl an Instruktionen in den Browsern Firefox und Chrome.

Das zweite Programm in Appendix B.2 berechnet den Binomialkoeffizienten  $\binom{n}{k}$ , indem es das Pascalsche Dreieck durchläuft ( $n$  entspricht Zeile des Dreiecks und  $k$  entspricht der Spalte des Dreiecks). Dieses Programm wurde gewählt, da es im Gegensatz zum ersten Programm wiederholt Lade/Speicherinstruktionen mit mehreren Registern verwendet. Es arbeitet außerdem viel mit dem Stack, da es oft Subroutinen aufgerufen werden, um in die nächste

Zeile des Dreiecks zu kommen. Die Anzahl an Instruktionen nimmt sehr schnell große Werte an, wenn man bereits kleine Zahlen für  $n$  und  $k$  nimmt (bereits bei  $n, k \geq 20$  mehrere Millionen Instruktionen). Das Ergebnis des Benchmarks für dieses Programm ist in Abbildung 22 illustriert. Daraus lassen sich eine durchschnittliche Ausführung von 374000 Instruktionen/s für Firefox und 506000 Instruktionen/s für Chrome berechnen. Mit Chrome lassen sich also 35.3% mehr solcher Instruktionen pro Sekunde ausführen als mit Firefox.

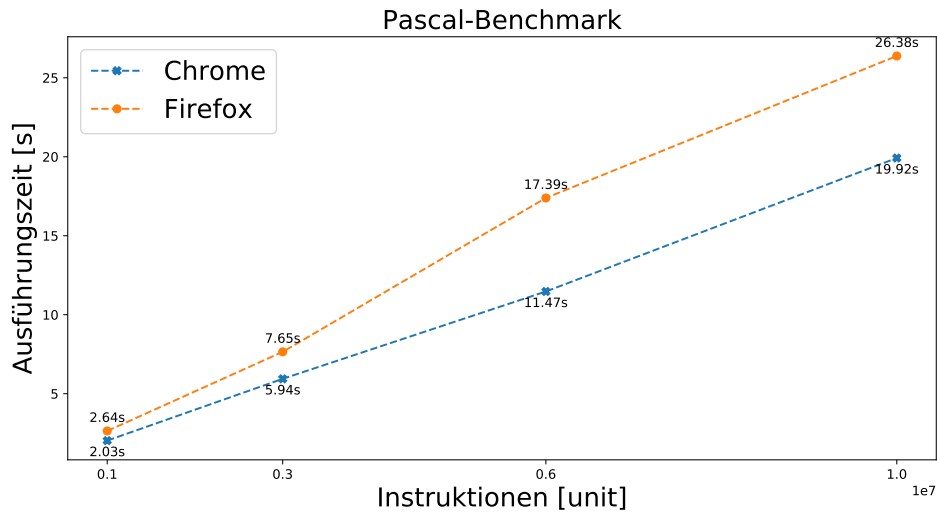


Abbildung 22: Benchmark-Test für das Programm zum Berechnen des Binomialkoeffizienten in Appendix B.2. Getestet wurde die Zeit (Durchschnitt von 5 Durchläufen) für eine bestimmte Anzahl an Instruktionen in den Browsern Firefox und Chrome.

Die Performance sollte leicht für die kleinen Programme aus den Proseminaren ausreichen. Anhand der Benchmarks lässt sich auch darauf schließen, dass unter Chrome schnellere Ausführungszeiten erreicht werden können.

## 5 Zusammenfassung

Der Simulator implementiert alle nötigen Teile einer ARMv5 Entwicklungsumgebung um Assembler Programme schreiben, debuggen und analysieren zu können. Das Frontend hilft dabei die darunterliegenden Prozesse bei Ausführung eines ARM-Programms zu verstehen. Die Performance der Webanwendung ist ausreichend für die kleinen Programme, die Student:innen für die Proseminare schreiben, und konnte an mehreren früheren Beispielen erfolgreich getestet werden.

Die Voraussetzungen aus dem Exposé (Implementation aller für die Vorlesung/das Proseminar benötigten ARMv5-Instruktionen, Webanwendung mit Anzeige von Registern, Stack und Teilen des Hauptspeichers, Standardfunktionen eines Debuggers und Testen der Funktionsweise an älteren Beispielen) konnten erfüllt werden. Die optionalen Ziele (Erstellen von Vorlagen für zukünftige Proseminar mit Überprüfung auf Korrektheit und automatische Code-Vervollständigung mit Hinweisen) sind sich leider zeitlich nicht mehr ausgegangen. Beispiele können zwar definiert und über ein Dropdown-Menü geladen werden, jedoch gibt es keine Funktionalitäten zur automatischen Überprüfung des Codes.

Der Code wurde in sinnvolle Klassen und Datentypen (Instruktionen und dazugehörige Operanden) eingeteilt um eine zukünftige Erweiterung so leicht wie möglich zu gestalten. Die erste nützliche Verbesserung wäre das Schreiben von Unittests, um bei Veränderung oder Erweiterung des Codes, die Korrektheit der Implementation beizubehalten. Bei der Implementation habe ich mich bemüht so viele Bugs wie möglich zu finden und zu beheben, habe jedoch sicherlich nicht alle gefunden. Falls Student:innen den Simulator in zukünftigen Proseminaren verwenden, finden diese bestimmt Bugs, die ich nicht bedacht habe, und zusätzliche Funktionalitäten, um die Bedingung des Simulators zu erleichtern oder verbessern.

## Literatur

- [1] ARM Limited. GNU Toolchain for ARM processors. Zugriffen am: 29.09.2021. <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain>.
- [2] ARM Limited. ARMv5 Architecture Reference Manual - Issue I, 2005.
- [3] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, 2014.
- [4] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
- [5] E. Davey. tsPEG: A PEG Parser Generator for TypeScript. Zugriffen am: 29.09.2021. <https://github.com/EoinDavey/tsPEG>.
- [6] Facebook. React. Zugriffen am: 29.09.2021. <https://reactjs.org/>.
- [7] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004.
- [8] P. Knaggs. ARM Assembly Language Programming, 2016.
- [9] L . Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1), January 2002.
- [10] Microsoft. TypeScript. Zugriffen am: 29.09.2021. <https://www.typescriptlang.org/>.
- [11] Microsoft. Windows Subsystem for Linux. Zugriffen am: 29.09.2021. <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.
- [12] The GNU Project. GDB: The GNU Project Debugger. Zugriffen am: 29.09.2021. <https://www.gnu.org/software/gdb/>.
- [13] The QEMU Project Developers. QEMU User Mode Emulation. Zugriffen am: 29.09.2021. <https://qemu.readthedocs.io/en/latest/user/index.html>.
- [14] H. Wong. CPULATOR: A CPU and I/O device simulator. Zugriffen am: 29.09.2021. <https://cpulator.01xz.net/?sys=arm>.

## A Parsing Expression Grammatik

```
// tsPEG doesn't allow all properties of regular expressions,
// so cannot use the one to ignore case. Therefore changed
// all direct word like '.text' to '.[tT][eE][xX][tT]' with
// a python script. Before tried to convert the user input
// to lower case, before parsing. But this messed with the
// .ascii content with upper case letters.

start := start=line

// grammar split into individual lines with 4 different
// contents in currentLine. '[\s]*' handles empty lines
// between lines with content
line := '[\s]*' label=label? '[\s]*' currentLine=directive wso
      comment=comment? nend nextLine=line |
      '[\s]*' label=label? '[\s]*' currentLine=instruction wso
      comment=comment? nend nextLine=line |
      '[\s]*' label=label? '[\s]*' currentLine=variableLine
      nend nextLine=line |
      '[\s]*' label=label? '[\s]*' currentLine=commentLine
      nend nextLine=line |
      '[\s]*' $

// 1. all directives the parser knows
directive := directive=ascii |
           directive=space |
           directive='.[aA][rR][mM]' |
           directive='.[tT][eE][xX][tT]' |
           directive='.[dD][aA][tT][aA]' |
           directive='.[aA][lL][iI][gG][nN]' |
           directive='.[gG][lL][oO][bB][aA][lL]' ws
           '_[sS][tT][aA][rR][tT]'

// directives where additional information need to be stored
ascii := '.[aA][sS][cC][iI][iI]' ws '"' data='[ -!#-~]*' '"'
space := '.[sS][pP][aA][cC][eE]' ws size='[0-9]+'

// 2. all instruction the parser knows further divided
// into instruction types
instruction := instruction=art |
              instruction=log |
              instruction=copyJump |
              instruction=loadStore |
              instruction=loadStoreMultiple |
              instruction=softwareInterrupt

// 3. lines with variables, like after ascii "len = . - msg
variableLine := variable='[_A-Za-z][_A-Za-z0-9]*' wso '=' wso
              '.' wso '-' wso label='[_A-Za-z][_A-Za-z0-9]*'

// 4. line with only a comment
commentLine := commentLine=comment

// optional label and comment for each line
label := label='[_A-Za-z][_A-Za-z0-9]*' ':'
comment := comment='//[ \t\S]*' | comment='\/\*[ \t - . 0 ~ ö ä ü ß ]*\*\/'

//-----
```

```

// arithmetic or multiplication instructions
art := inst=artInst cond=condition ws operands=artOp |
      inst='[mM][uU][lL]' cond=condition ws operands=artMulOp |
      inst='[mM][lL][aA]' cond=condition ws operands=artMlaOp

// known arithmetic instructions
artInst := '[aA][dD][dD]' | '[aA][dD][cC]' | '[sS][uU][bB]' |
            '[sS][bB][cC]' | '[rR][sS][bB]' | '[rR][sS][cC]'

// different kinds of operands for arithmetic instructions
artOp := artOp3 | artOp2
artOp2 := op1=regOp wso ',' wso op2=op
artOp3 := op1=regOp wso ',' wso op2=regOp wso ',' wso op3=op

// different kinds of operands for multiplication instructions
artMulOp := op1=regOp wso ',' wso op2=regOp wso ',' wso op3=regOp
artMlaOp := op1=regOp wso ',' wso op2=regOp wso ',' wso op3=regOp
           wso ',' wso op4=regOp

//-----

// logic instructions
log := inst=logInst cond=condition ws operands=logOp |
      inst=logCmpInst cond=condition ws operands=logOp2

// known logic instructions
logInst := '[aA][nN][dD]' | '[oO][rR][rR]' |
            '[eE][oO][rR]' | '[bB][iI][cC]'
logCmpInst := '[cC][mM][pP]' | '[cC][mM][nN]' |
              '[tT][sS][tT]' | '[tT][eE][qQ]'

// different kinds of operands for logic instructions
logOp := logOp3 | logOp2
logOp2 := op1=regOp wso ',' wso op2=op
logOp3 := op1=regOp wso ',' wso op2=regOp wso ',' wso op3=op

//-----

// copy or jump instructions
copyJump := inst=copyInst cond=condition ws operands=copyOp |
            inst=jumpInst1 cond=condition ws operands=jumpOp |
            inst=jumpInst2 cond=condition ws operands=jumpOp

// known copy or jump instructions
copyInst := '[mM][oO][vV]' | '[mM][vV][nN]'
jumpInst1 := '[bB]'
jumpInst2 := '[bB][lL]'

// different kinds of operands for copy or jump instructions
copyOp := op1=regOp wso ',' wso op2=op
jumpOp := op1=branchOp

//-----

// load/store/swap instructions
loadStore := inst=loadStoreInst format=format cond=condition ws
             operands=loadStoreOp |
             inst='[sS][wW][pP]' format=format cond=condition ws
             operands=swpOp |

```

```

        inst=loadStoreInst cond=condition ws
        operands=loadImmediateBranchOp |
        inst=loadStoreInst cond=condition ws
        operands=loadImmediateOp

// known load/store instructions and formats
loadStoreInst := '[lL][dD][rR]' | '[sS][tT][rR]'
format := '[bB]' | '[hH]' | '[sS][bB]' | '[sS][hH]' | ''

// different kinds of operands for load/store/swap instructions
loadStoreOp := op1=regOp wso ',' wso op2=addressingMode
swpOp := op1=regOp wso ',' wso op2=regOp wso ',' wso
        '\[' wso op3=regOp wso '\]'
loadImmediateOp := op1=regOp wso ',' wso op2=immOp
loadImmediateBranchOp := op1=regOp wso ',' wso '=' op2=branchOp
        offset='[+-][0-9]+'?

//-----

// load/store multiple instructions
loadStoreMultiple := inst=loadStoreMultipleInst
        addressingMode=loadStoreMultipleAddrMode
        cond=condition ws operands=loadStoreMultipleOp

// known load/store multiple instructions and format
loadStoreMultipleInst := '[lL][dD][mM]' | '[sS][tT][mM]'
loadStoreMultipleAddrMode := '[fF][dD]' | '[fF][aA]' | '[eE][dD]' |
        '[eE][aA]' | '[iI][aA]' | '[iI][bB]' |
        '[dD][aA]' | '[dD][bB]' | ''

// different kinds of operands for load/store multiple instructions
loadStoreMultipleOp := op1=regOp increment='!'? wso ',' wso
        '{' wso op2=regOpList wso '}'

//-----

// software interrupts
softwareInterrupt := inst='[sS][wW][iI]' cond=condition ws
        operands='#0'

//-----

// all operands for the instructions

// combinations of allows register, immediate and shifter operands
// for data-processing instructions
op := shiftOp=shiftOp | regImmOp=regImmOp
regImmOp := regOp=regOp | immOp=immOp
shiftOp := opToShift=regImmOp wso ',' wso shiftType=shiftType wso
        opShift=regImmOp

// addressing operands for load/store
addressingMode := '\[' wso reg=regOp wso '\]' offset=offset? |
        '\[' wso reg=regOp offset=offset? wso '\]'
        increment='!'?

offset := wso ',' wso sign=sign offset=op

// register operand
regOp := regOp='[rR][0-9]+' | '[pP][cC]' | '[lL][rR]' | '[sS][pP]'

```

```

// immediate operand
immOp := immType=immType sign=sign base=base number='[0-9a-fA-F]+'
immType := '#' | '='
base := '0[xX]' | '0[bB]' | '0[oO]' | ''
sign := '-' | '\+' | ''

// branch operand
branchOp := '[_A-Za-z][_A-Za-z0-9]*'

// list of register operands for load/store multiple
regOpList := op=regOpOrRange wso ',' wso nextOp=regOpList |
              op=regOpOrRange
regOpRange := op1=regOp wso '-' wso op2=regOp
regOpOrRange := op=regOpRange | op=regOp

// known shift types
shiftType := '[lL][sS][lL]' | '[aA][sS][lL]' | '[lL][sS][rR]' |
              '[aA][sS][rR]' | '[rR][oO][rR]' | '[rR][rR][xX]'

// conditions and S flag
condition := condType=conditionType updateStatusReg='[sS]'?

// known condition types
conditionType := '[eE][qQ]' | '[nN][eE]' | '[hH][sS]' | '[cC][sS]' |
                  '[lL][oO]' | '[cC][cC]' | '[mM][iI]' | '[pP][lL]' |
                  '[vV][sS]' | '[vV][cC]' | '[hH][iI]' | '[lL][sS]' |
                  '[gG][eE]' | '[lL][tT]' | '[gG][tT]' | '[lL][eE]' |
                  '[aA][lL]' | '[nN][vV]' | ''

//-----

// white space no new line
ws := '[\t]+'
// optional white space no new line
wso := '[\t]*'
// new line or end of file
nend := '\n' | '$'

```

## B Benchmark Code

### B.1 Division

```

_start:
    LDR r0, =1024    // Dividend
    LDR r1, =245     // Divisor
    MOV r2, #0       // Setze Hilfsregister auf 0
    BL div32         // Division 32 Bit

div32:
    MOV r3, #32     //Schleife

div32loop:
    MOVS r0, r0, LSL #1    // Höchstes Bit in C und niedrigstes Bit 0
    MOV r2, r2, LSL #1     // Verschiebe Hilfsregister um 1 Bit
    ADC r2, r2, #0         // Addiere den Carry-Bit auf niedrigstes Bit

    RSBS r2, r1, r2        // Subtrahiere Divisor von Hilfsregister

```



```

    ORRPL r0, r0, #1      // positiv - Setze niedrigstes Bit von r0 auf 1
    ADDMI r2, r2, r1      // negativ - Wiederherstellung des Rests
    SUBS r3, r3, #1
    BNE div32loop
    B _start

```

## B.2 Binomialkoeffizient

```

.arm
.text
.global _start

_start:
    LDR r0, =13 // n
    LDR r1, =7  // k
    BL pas      // Routine für Pascal-Loop

    MOV r1, r0  // Wert nach r1 kopieren für dec Ausgabe
    BL dec      // Dezimal Ausgabe von vorigem Blatt

    MOV r0, #0  // exit syscall
    MOV r7, #1
    SWI #0

pas:
    STMFID sp!, {r2-r12, lr} // Register sichern

    CMP r1, #0      // Vergleiche k mit 0
    MOVEQ r0, #1    // Wenn k = 0, ist der Wert...
    BEQ rec_end     // ...an dieser Stelle 1
    MOVLTI r0, #0   // Wenn k < 0, wird der Wert...
    BLT rec_end     // ...mit 0 initialisiert

    CMP r1, r0      // Vergleiche k mit n
    MOVEQ r0, #1    // Wenn k = n, ist der Wert...
    BEQ rec_end     // ...an dieser Stelle 1
    MOVGT r0, #0    // Wenn k > n, wird der Wert...
    BGT rec_end     // ...mit 0 initialisiert

    CMP r0, #1      // Vergleiche n mit 1
    MOVLE r0, #1    // Wenn n <= 1, ist der Wert...
    BLE rec_end     // ...an dieser Stelle 1

    MOV r4, r0      // Werte von n und k werden...
    MOV r5, r1      // ...nach r4 und r5 kopiert
    SUB r0, r4, #1   // n - 1
    SUB r1, r5, #1   // k - 1
    BL pas

    MOV r2, r0      // Wert erster Summand nach r2 kopiert
    MOV r3, r1
    SUB r0, r4, #1   // n - 1
    MOV r1, r5       // k
    BL pas

    ADD r0, r2      // Addition (n-1 / k-1) + (n-1 / k)...
                    // ...von rekursiver Formel wird ausgeführt

rec_end:

```

```

    LDMFD sp!, {r2-r12, lr} // Gespeicherte Register wiederherstellen
    MOV pc, lr              // Rücksprung

dec:                                // Ganzzahl in r1 (32 Bit, vorzeichenlos)
    STMFD sp!, {r0-r12, lr} // alle Register sichern
    LDR r5,=bufferdec+10     // Zeiger auf Ende des Puffers +1
    MOV r6, #0x30            // ASCII-Kode für 0 als Offset
    MOV r0, #1               // wähle stdout
    MOV r7, #0               // Stellenzähler

decloop:
    ADD r7, r7, #1           // nächste Ziffer (mind. eine)
    MOV r2, #10              // Basis 10 (dezimal)
    BL div                   // r1 : r2 von Folie 5
    ADD r4, r6, r1, LSR #16  // Rest als Ziffer in ASCII . . .
    STRB r4, [r5,-r7]        // . . . rückwärts in Puffer schreiben
    BICS r1, r1, #0x000f0000 // Rest löschen
    BNE decloop              // mehr Stellen wenn Quotient > 0
    SUB r1, r5, r7            // Start der Zeichenkette im Puffer
    MOV r2, r7               // Länge der Zeichenkette
    MOV r7, #4               // Systemaufruf write wählen
    SWI #0

    LDR r1, =lb
    MOV r0, #1
    MOV r2, #1
    SWI #0

    LDMFD sp!, {r0-r12, lr} // alle Register wiederherstellen
    MOV pc, lr              // Rücksprung

div: // Dividend in r1 (16 Bit, vorzeichenlos)
    // Divisor in r2 (16 Bit, vorzeichenlos)

    MOV r2, r2, LSL #16
    MOV r3, #16              // Schleifenzähler

divloop:
    RSBS r1, r2, r1, LSL #1 // schiebe und subtrahiere
    ORRPL r1, r1, #1
    ADDMI r1, r1, r2          // Wiederherstellung des Rests
    SUBS r3, r3, #1
    BNE divloop

    // Quotient in r1_15, . . . , r1_0
    // Rest in r1_31, . . . , r1_16
    MOV pc, lr              // Rücksprung

lb: .ascii "\\n"            // Zeilenumbruch

.data
bufferdec: .space 10        // 10 Byte, denn log10(232) = 10

```