

ARM Simulator, Interpreter und Debugger als Webanwendung

Finalpräsentation

Zangerl Dominik

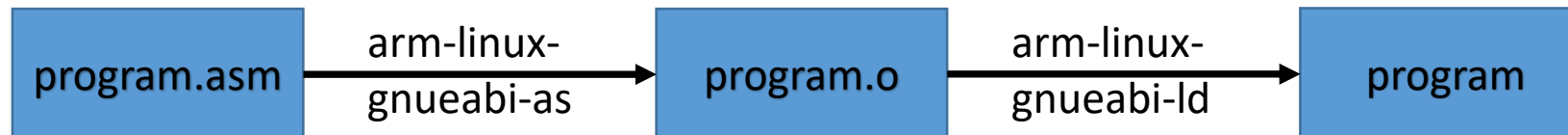
Betreuer: Alexander Schlögl

Gliederung

- Einleitung und Motivation
- Theorie
 - ARMv5
 - Parser
- Implementation
- Evaluation und Zusammenfassung
- Referenzen

ARMv5 im ersten Semester

- ARMv5 [\[2\]](#) als Beispiel einer Befehlssatzarchitektur
- Schreiben von Assembler-Programme und Ausführung auf einer ARMv5 Architektur
- Simulation mit GNU Toolchain [\[1\]](#):



- Ausführen mit QEMU User-Space-Emulator [\[13\]](#)
- Vereinfachung mit Skript und Ausführung über virtuelle Maschine oder WSL [\[11\]](#)

Debugging

- Größter Zeitaufwand bei Fehlersuche im Programm
- Kann zusammen mit dem Gnu Debugger [\[12\]](#) verwendet werden:

```
Register group: general
r0      0xfdbbf7af      -38013009      r1      0xfdffffe5      -33554459
r2      0x3             3              r3      0x4             4
r4      0x7             7              r5      0xffdffd4        -2097196
r6      0x4c4d5b53      1280138067     r7      0x8410de10      -2079269360
r8      0x37feffff      939458558      r9      0xffedfffc      -1179652
r10     0xfbaad45e      -72690594      r11     0x88cad3c4       -1999973436
r12     0xfdffffff      -33882113      sp      0x0             0x0
lr      0xffffffff      -1             pc      0x1c          0x1c <_start>
xPSR    0x1000000       16777216
```

Abbildung: Use GDB on an ARM assembly program [\[15\]](#)

- Arbeiten mit Debuggern im ersten Semester oft schwierig
- Großer Zeitaufwand zusammen mit Aufsetzen der Toolchain

ARMv5 Umgebung und Debugging als Webanwendung

- Bachelorprojekt: Simuliere ARMv5 Entwicklungsumgebung und Debugger als Webanwendung
- ARMv5 Entwicklungsumgebung
 - Simulierte CPU und Hauptspeicher
 - Assembler-Code in Webanwendung schreiben und direkt im Browser ausführen
 - Dauerhafte Anzeige von Registern und Stacks
- Debugger
 - Breakpoints
 - Zeilenweise Abarbeitung

ARMv5 Architektur

- Rechner mit reduziertem Befehlssatz (RISC [\[4\]](#)):
 - Load/Store-Architektur
 - Datenverarbeitende Instruktionen arbeiten nur mit Inhalten der Register
 - Einheitliche Form und Länge der Kodierung
- Zusätzlich bei ARM:
 - Meiste Instruktionen haben Zugriff ALU und Barrel-Shifter
 - Adressierungsarten, die Adresse automatisch inkrementieren/dekrementieren
 - Instruktionen zum Laden/Speichern von mehreren Registern

ARMv5 Register

- 31 Universalregister mit einer Breite von 32 Bit
 - 16 sichtbar, je nach Ausführungsmodus
 - R13 – Stapelzeiger
 - R14 – Link-Register
 - R15 – Befehlszähler
- Status-Register (CPSR) mit Status-Flags NZCV:
 - N – Negativ
 - Z – Null (Zero)
 - C – Übertrag (Carry)
 - V – Überlauf (Overflow)

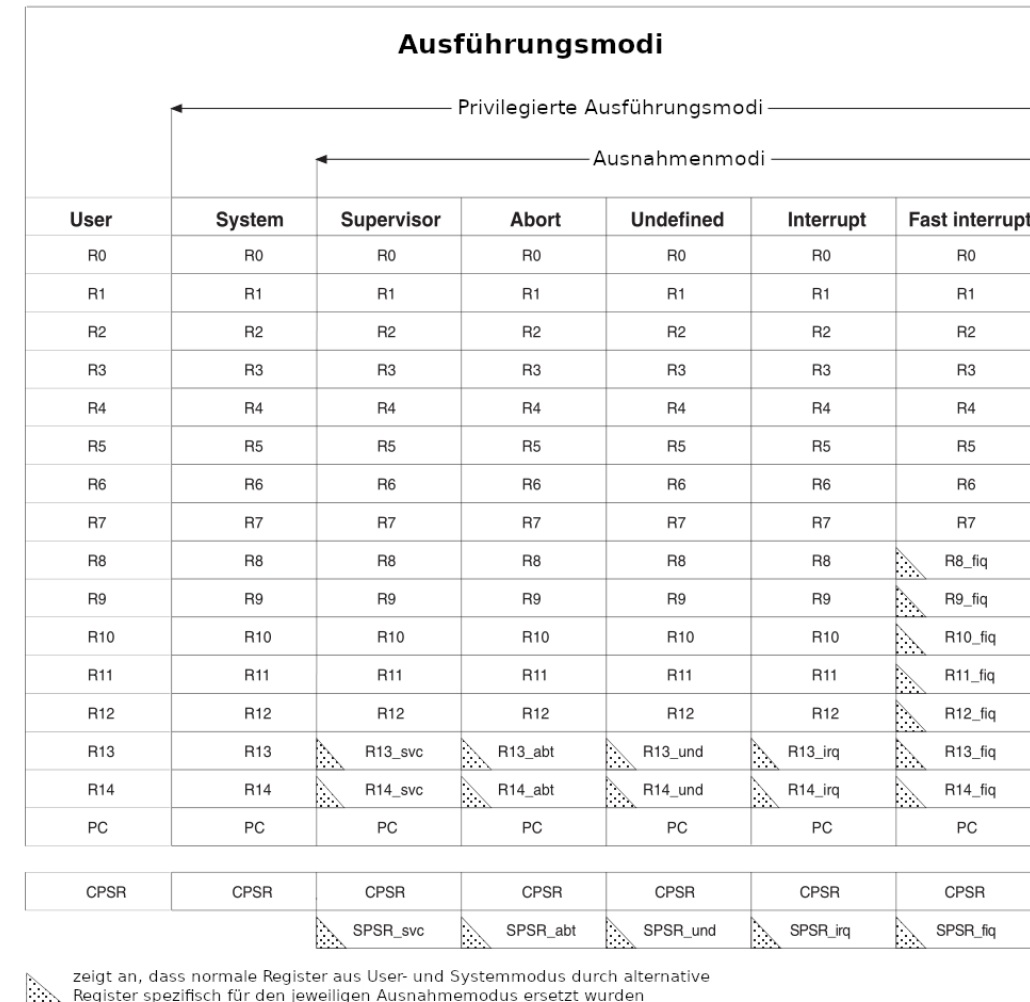


Abbildung: Register je nach Ausführungsmodi [2]

Bedingungen

- Instruktionen können nur unter bestimmten Bedingungen ausgeführt werden
- Höchste 4 Bits reserviert für Kodierung der Bedingung
- Geknüpft an die Status-Flags NZCV, werden vor Ausführung der Bedingung überprüft
- Mnemonik an Instruktion im Code anhängen

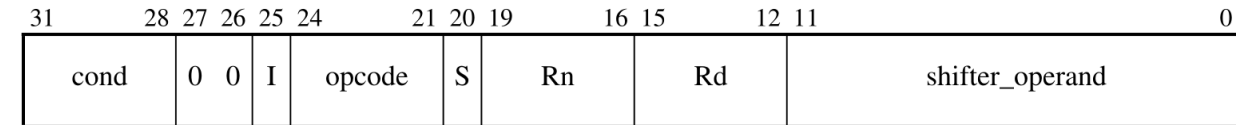


Kodierung [31:28]	Mnemonik	Bedeutung	Status-Flags
0000	EQ	Gleichheit (Equal)	Z=1
0001	NE	Ungleichheit (Unequal)	Z=0
0010	CS/HS	Carry-Bit gesetzt (Carry set)/ Vorzeichenlos größer oder gleich (unsigned higher or same)	C=1
0011	CC/LO	Carry-Bit nicht gesetzt (Carry clear)/ Vorzeichenlos kleiner (unsigned 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

Abbildungen: Beispiele für Bedingungen und Kodierung [\[2\]](#)

Datenverarbeitende Instruktionen

- Arithmetische- und Vergleichsoperationen
- Instruktionen über Befehlscode unterschieden
- Register-Operanden Rn und Rd
- Flexibler dritter Operand, der Zugriff auf Barrel-Shifter hat
- S-Bit aktualisiert die Status-Flags



Befehlscode [24:21]	Mnemonik	Operation	Aktion
0000	AND	Logisches Und	$Rd := Rn \text{ AND } shift_op$
0001	EOR	Logisches exklusives Oder	$Rd := Rn \text{ EOR } shift_op$
0010	SUB	Subtraktion	$Rd := Rn - shift_op$
0011	RSB	Umgekehrte Subtraktion	$Rd := shift_op - Rn$
0100	ADD	Addition	$Rd := Rn + shift_op$
0101	ADC	Addition mit Carry	$Rd := Rn + shift_op + C$
0110	SBC	Subtraktion mit Carry	$Rd := Rn - shift_op - \text{NOT}(C)$
0111	RSC	Umgekehrte Subtraktion mit Carry	$Rd := shift_op - Rn - \text{NOT}(C)$
1000	TST	Test	$Rn \text{ AND } shift_op$ und aktualisiere Flags
1001	TEQ	Äquivalenztest	$Rn \text{ EOR } shift_op$ und aktualisiere Flags

Abbildungen: Datenverarbeitende Instruktionen [2]

Shifter-Operand

- Immediate-Wert (8 Bit + Rotation)
- Register
- Verschiebeoperation
 - Logische Linksverschiebung
 - Arithmetische/Logische Rechtsverschiebung
 - Rechtsrotation
 - Erweiterte Rechtsrotation

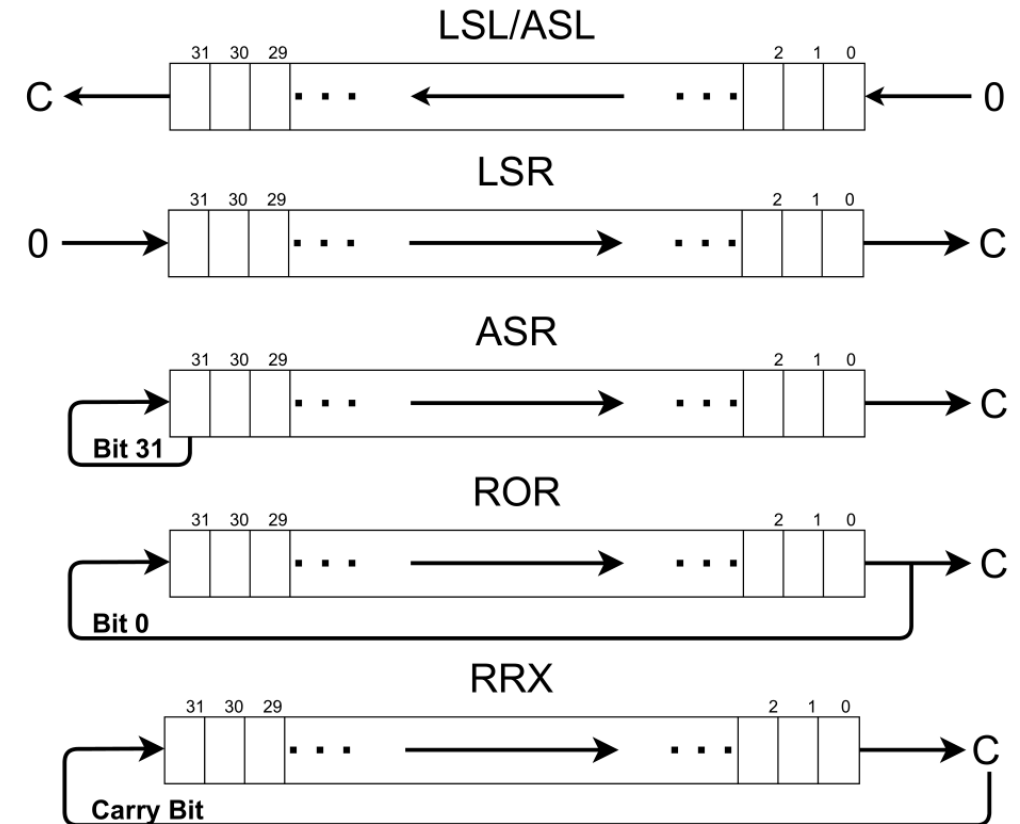


Abbildung: Verschiebeoperationen [8]

Sprunginstruktionen

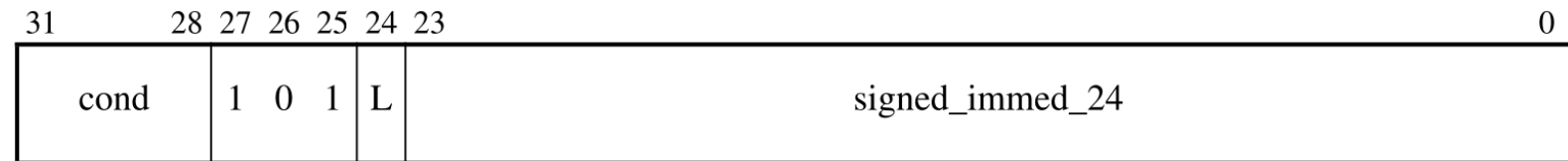


Abbildung: Kodierung Sprungoperation [\[2\]](#)

- Adressen im Hauptspeicher können Labels zugewiesen werden
- Operationen können Sprünge zu diesen Labels durchführen
 - Adresse des Labels wird in Register für Befehlszähler kopieren
 - Abstand zu aktueller Adresse berechnet und in *signed_immed_24* gespeichert
 - L-Bit zum Hinterlegen der Rücksprungadresse im Link-Register

Lade- und Speicherinstruktionen

- Lädt Inhalt von Adresse im Hauptspeicher in Zielregister oder speichert Inhalt eines Registers in den Hauptspeicher
- Auch Laden/Speichern von Halbwörtern und Bytes möglich
- Adressierungsart:
 - Offset
 - Pre- oder Post-indexed
 - Auto-Inkrement

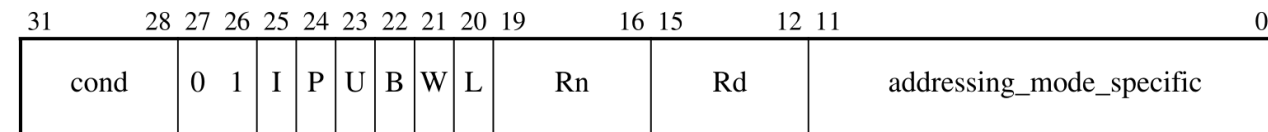


Abbildung: Kodierung Lade- und Speicherinstruktionen [2]

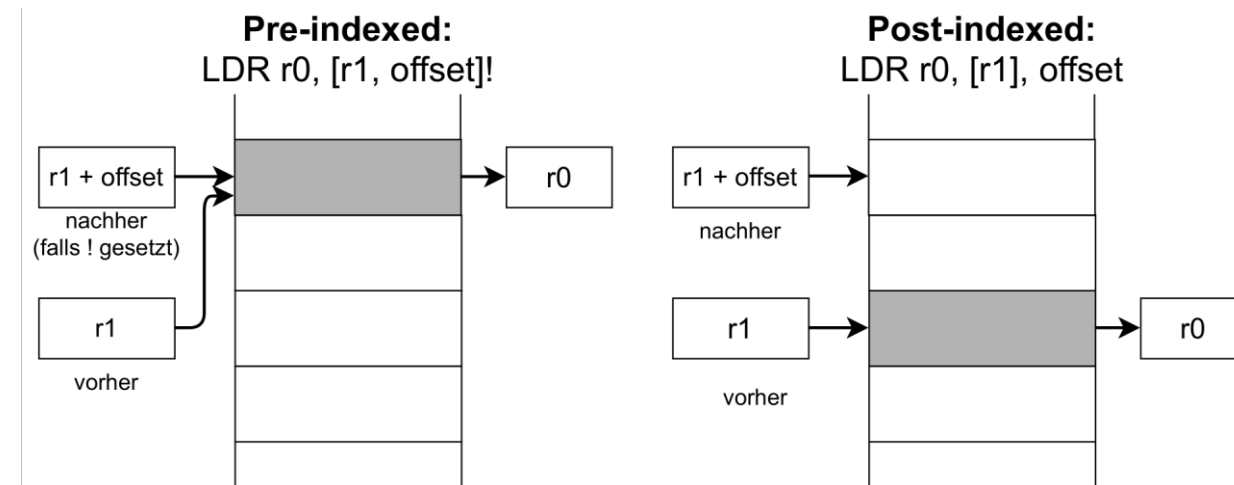


Abbildung: Adressierungsarten [8]

Lade- und Speicherinstruktionen für mehrere Register

- Register werden als Liste angegeben und dann ausgehend vom Basisregister Rn geladen oder geschrieben
- Adressierungsarten **IA/IB/DA/DB**:
 - I**ncrement/**D**ecrement: Basisregister wird erhöht/verringert
 - A**fter/**B**efore: Aktualisierung des Basisregister vor/nach Laden oder Speichern
- Alternative Stack-Adressierungsarten **FA/FD/EA/ED**:
 - A**scending/**D**escending: Basisregister wird erhöht/verringert
 - F**ull/**E**mpy: Stapelzeiger weist auf gefüllte/leere Adresse

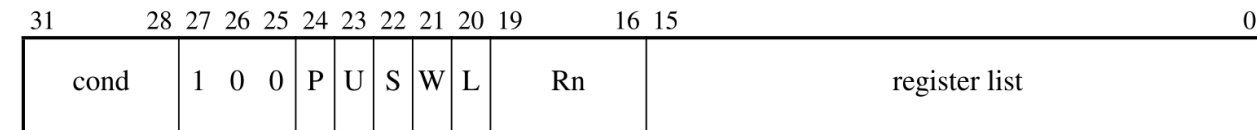


Abbildung: Kodierung Lade- und Speicherinstruktionen für mehrere Register [2]

Parsing Expression Grammatik

CFG

```
1. A --> a | a b
2. A --> a b | a
```

PEG

```
1. A <-- a / a b
2. A <-- a b / a
```

Abbildung: Vergleich kontextfreie Grammatik und Parsing Expression Grammatik [7]

- Ungeordneter Alternativen-Operator | bei kontextfreien Grammatiken
 - Ungeordnet – Definition 1 und 2 gleichwertig
 - Mehrdeutigkeiten – Parser muss alle Alternativen betrachten
- Alternativen-Operator mit Priorität / bei Parsing Expression Grammatiken
 - Alternativen mit absteigender Priorität geordnet
 - Effizienter – Parser kann nach gefundenem Match stoppen

tsPEG

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

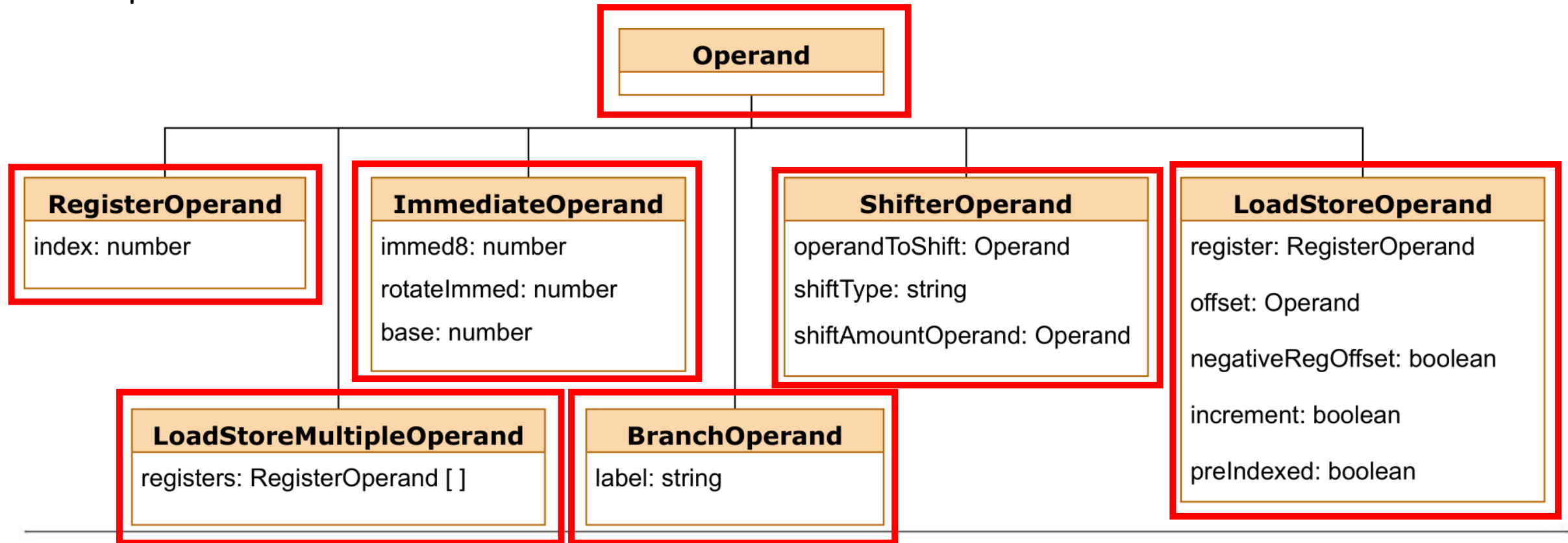
Abbildung: Hello World Beispiel für tsPEG [\[5\]](#)

- tsPEG [\[5\]](#) ist ein Parser-Generator für TypeScript [\[10\]](#)
 - Definiere Regeln von PEGs mit `:=`
 - Zuweisen von Variablen mit `=`
 - Reguläre Ausdrücke oder Strings zum Matchen
- Erfolgreiches Parsen:
 - Generiert abstrakten Syntax-Baum aus zugewiesenen Variablen in TypeScript

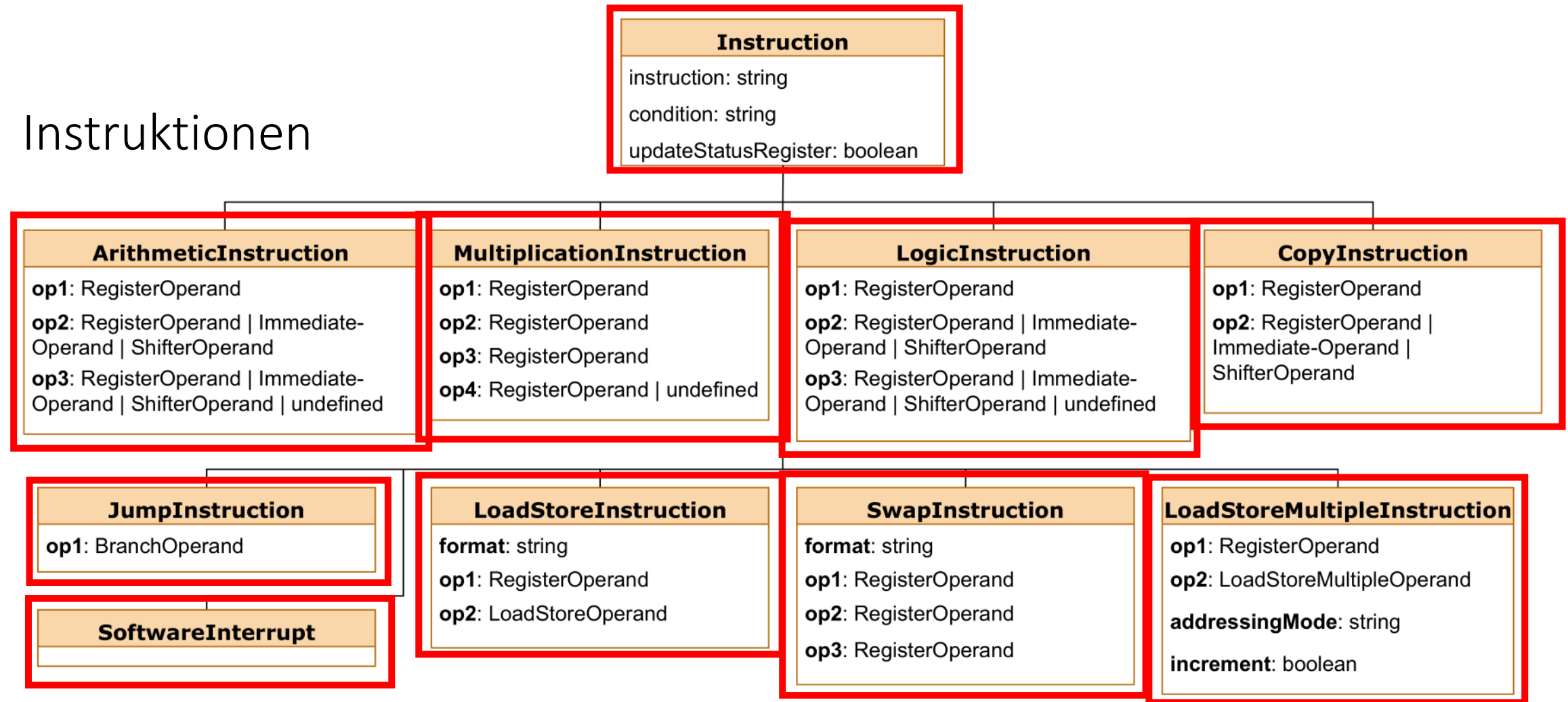
Aufbau

- Klassen für Operanden
 - Instruktionen aufgebaut aus Operanden
- Einteilung:
 - CPU
 - Parser
 - Hauptspeicher
 - Code-Ausführung

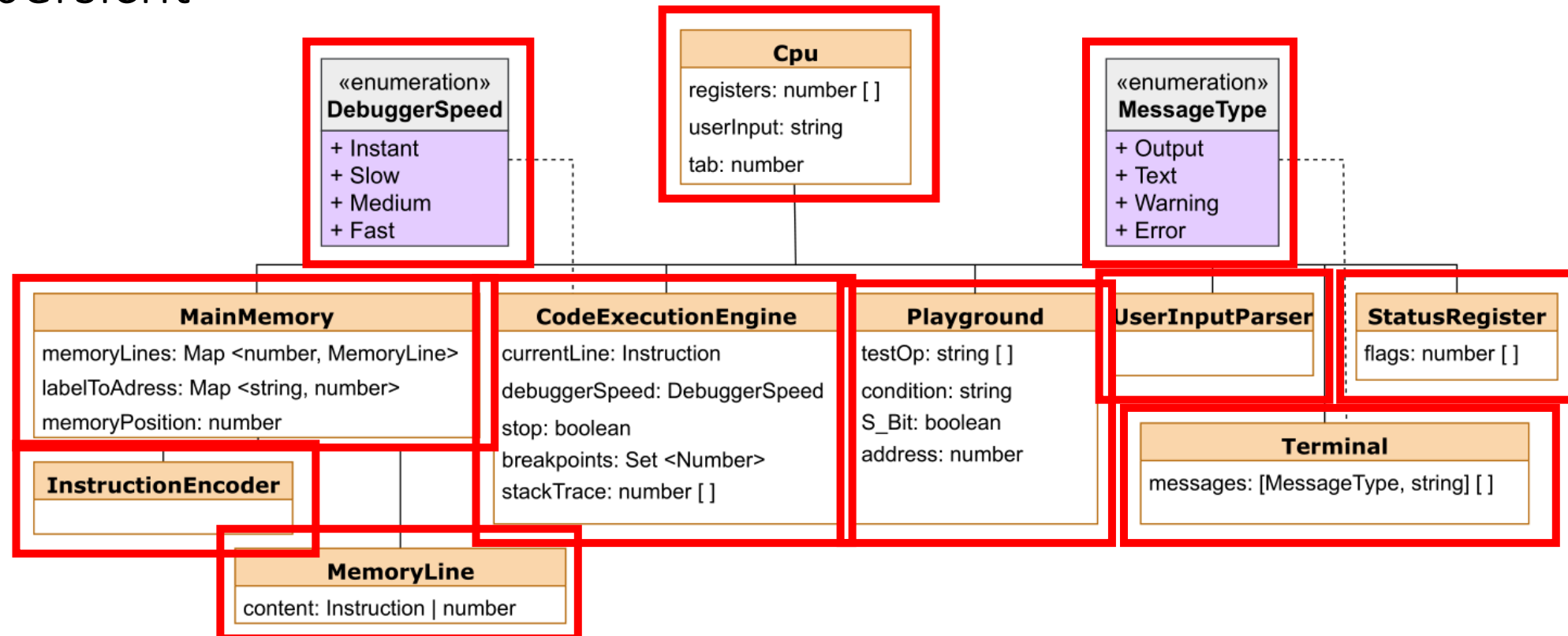
Operanden



Instruktionen



Übersicht



Parser

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

- Einteilung der Grammatik in Zeilen:
 - Direktiven
 - Instruktionen
 - Kommentare
- Optionales Label und Kommentar
- Jede Zeile besitzt weitere Zeile, außer das Ende des Codes \$ ist erreicht

Parser

- Weitere Einteilung je nach Art der Zeile
- Instruktionen aufgeteilt in Typen
- Instruktion:
 - Name der Instruktion
 - Bedingung
 - Operanden

```
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
```

Parser

- Abarbeiten des abstrakten Syntax Baums
Zeile für Zeile
- Zuweisen der aktuellen Zeile gefolgt von
While-Loop:
 - Aufrufen der korrekten Funktion
 - Zuweisen der nächsten Zeile

```
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 }
```

Hauptspeicher

- Klasse mit Funktionen zum Hinzufügen von Instruktionen, Daten und Labels mit Überprüfung auf Korrektheit
- Anzeige des Hauptspeichers:
 - Breakpoints
 - Adresse
 - Kodierung bzw. Daten
 - Instruktion
- Hervorheben der aktuellen Instruktion

●	Address	Encoding	Instruction
		msg:	
	00000000	6c6c6548	
	00000004	6e49206f	
	00000008	7262736e	
	0000000c	216b6375	
	00000010	0000000a	
		_start:	
	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

Code Execution Engine

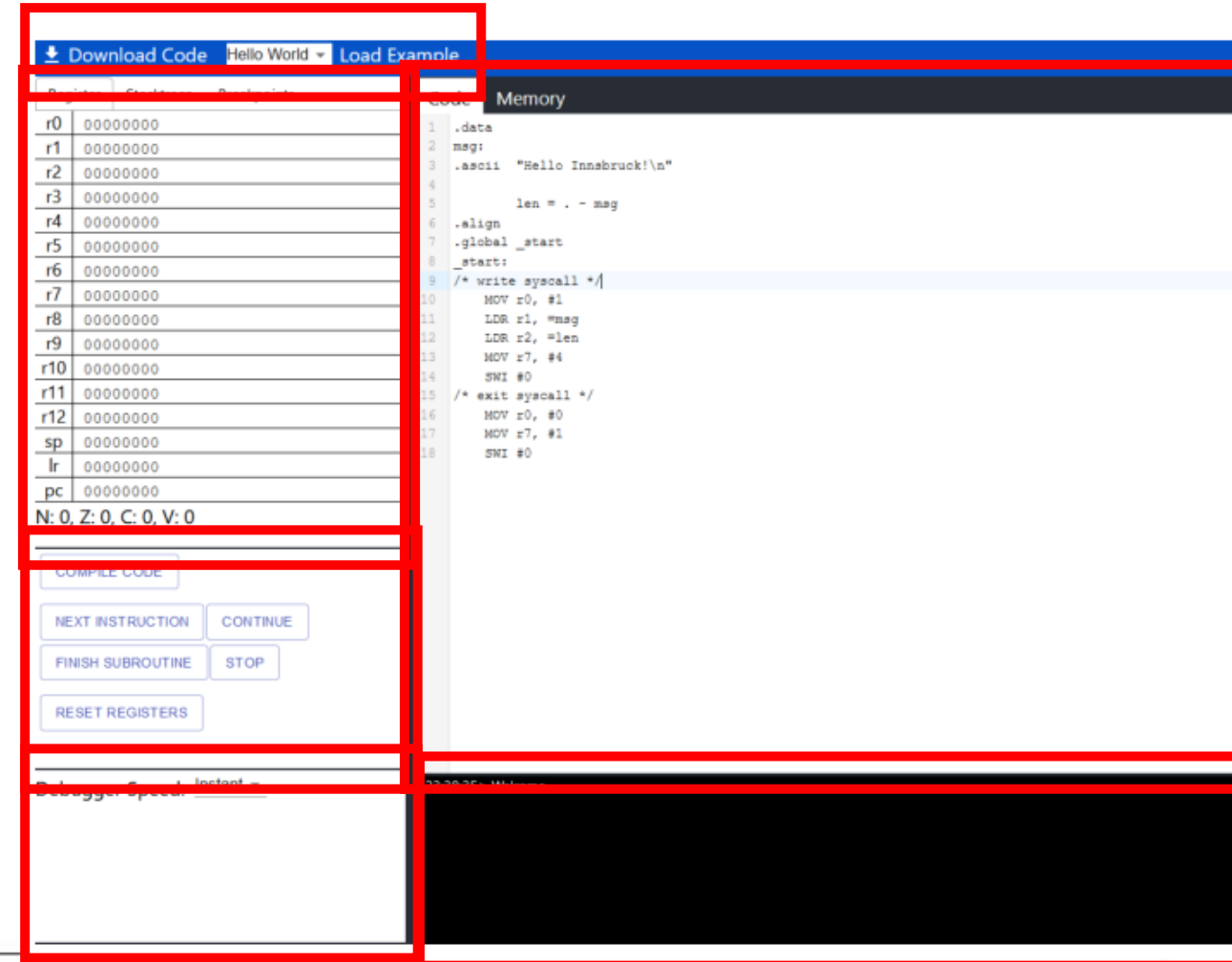
- Klasse zur Ausführung der Instruktionen im Hauptspeicher
- Aufgeteilt in 3 Hauptfunktionen:
 - 1. *continue()*
 - Asynchrone Funktion mit unterschiedlichen Debugger-Geschwindigkeiten
 - Überprüft Abbruchbedingungen (Breakpoints, Ende einer Subroutine)
 - Ruft Funktion zum Ausführen der nächsten Instruktion auf
 - Aktualisierung der Benutzeroberfläche

Code Execution Engine

- 2. *executeNextInstruction()*
 - Überprüft korrekt ausgerichtete Adresse
 - Holt nächste Instruktion aus Hauptspeicher
 - Aktualisiert Stacktrace
- 3. *executeInstruction()*
 - Überprüft Ausführungsbedingung
 - Ruft korrekte Funktion, je nach Typ der Instruktion auf
 - Aktualisiert Register, Hauptspeicher

Benutzeroberfläche

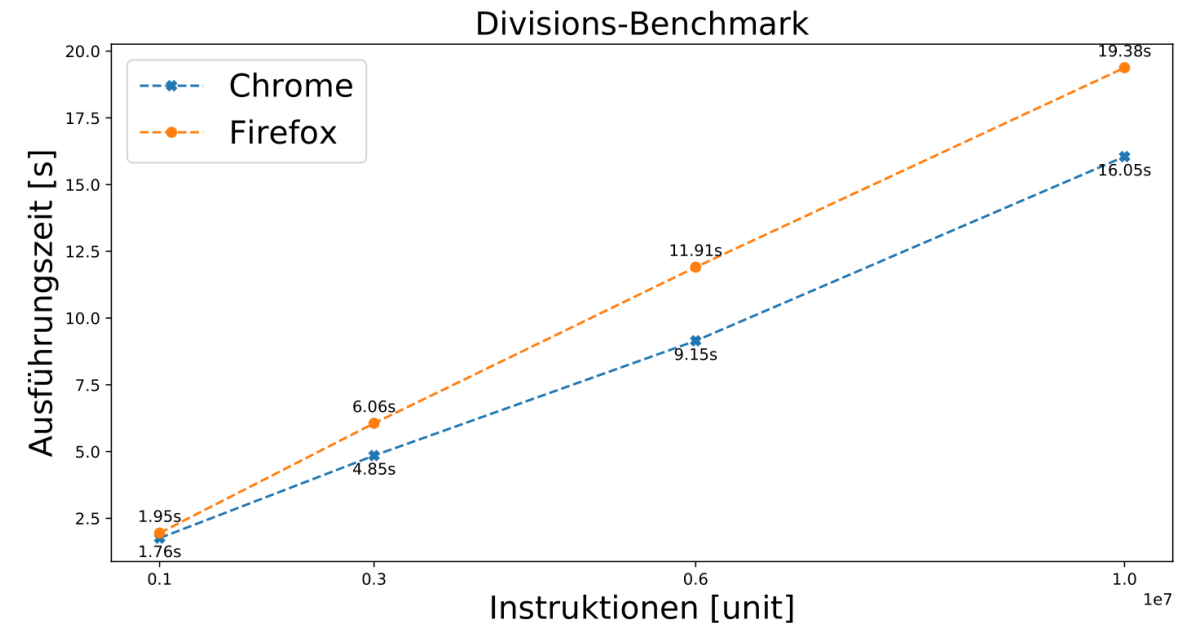
- Header mit Download-Button und Dropdown-Menü zum Laden von Beispielen
- Aktuelle Register, Stacktrace und Breakpoints
- Debugger
- Optionen
- Feld für Benutzereingabe
- Terminal



Live Demo

Evaluation

- Korrektheit getestet mit Beispielen aus Vorlesung/Proseminar – Verfügbar über Dropdown-Menü in der Webanwendung
- Benchmark für Ausführungszeit (i5-4450 @ 3.20 GHz):
 - Arithmetische Instruktionen (Divisions-Beispiel)
 - 507000 Inst/s (Firefox) bzw. 616000 Inst/s (Chrome)
 - Lade- und Speicherinstruktionen (Pascal-Beispiel)
 - 374000 Inst/s (Firefox) bzw. 506000 Inst/s (Chrome)



Zusammenfassung

- Simulator mit allen nötigen Teilen einer ARMv5 Entwicklungsumgebung um Assembler Programme schreiben, debuggen und analysieren zu können
- Ausreichende Performance für die kleinen PS-Programme
- Aufteilung in Operanden, Instruktionen und Teile einer CPU um Erweiterung durch zusätzliche Funktionen zu erleichtern

Referenzen

- [1] ARM Limited. GNU Toolchain for ARM processors. Zugegriffen 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. Zugegriffen am: 29.09.2021. <https://github.com/EoinDavey/tsPEG>.
- [6] Facebook. React. Zugegriffen 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. Zugegriffen am: 29.09.2021 . <https://www.typescriptlang.org/>.
- [11] Microsoft. Windows Subsystem for Linux. Zugegriffen am: 29.09.2021. <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.
- [12] The GNU Project. GDB: The GNU Project Debugger. Zugegriffen am: 29.09.2021. <https://www.gnu.org/software/gdb/>.
- [13] The QEMU Project Developers. QEMU User Mode Emulation. Zugegriffen am: 29.09.2021. <https://qemu.readthedocs.io/en/latest/user/index.html>.
- [14] H. Wong. CPULator: A CPU and I/O device simulator. Zugegriffen am: 29.09.2021. <https://cpulator.01xz.net/?sys=arm>.
- [15] J. Mossberg. Use GDB on an ARM assembly program. Zugegriffen am: 04.03.2021. <https://jacobmossberg.se/posts/2017/01/17/use-gdb-on-arm-assembly-program.html>

