Konzeption und Simulation einer fiktiven 32-Bit-Architektur sowie die Betrachtung gängiger Sortieralgorithmen auf jener

Jonathan Frech
(`info@jfrech.com`)

22. August bis 14. November 2020

1 Architektur

Zu Lehrzwecken im Wintersemester 2020 habe ich eine möglichst einfach gehaltene 32-Bit-Architektur konzipiert sowie einen Interpreter für jene geschrieben. Die Architektur ist einer RAM-Maschine nachempfunden und besitzt neben einem kontinuierlichen unausgerichteten Speicherblock vier Register; zwei davon – Register A und B – werden zur Berechnung von Werten sowie Speicheradressen benutzt. Ein weiteres Register P bildet einen Programmzähler ab und S einen Stapelzeiger.
Mittels dieser Architektur werden im Folgenden zwei Sortieralgorithmenfamilien untersucht: Quicksort (siehe 3.1) und Shellsort (siehe 3.2).

Der Quelltext des Interpreters samt Testskript kann unter

`https://github.com/jfrech/Joy-Assembler/releases/tag/v1.0`

eingesehen werden (circa 2 500 Zeilen `C++`). Die Kompilation wurde erfolgreich mit den Compilern `g++ 9.3.0`, `g++ 10.2.0`, `clang 10.0.0`, sowie `Apple clang 11.0.0` und `Microsoft® C/C++ Optimizing Compiler Version 19.27.29111 for x86` durchgeführt.

2 Simulation

Der Interpreter simuliert die fiktive Maschine oben beschriebener Architektur mittels sukzessiver Berechnung jeden Rechenschritts. Somit können schrittweise die Ausführung betrachtet und exakte Laufdauerparameter erhoben werden.
Um die Güte eines Sortierverfahrens zu testen gibt es eine Vielzahl an Parameterkandidaten. Im Folgenden wird die *Ausführungsgeschwindigkeit* betrachtet, genauer die Anzahl der benötigten Instruktionen, gewichtet nach ihrer Mikroinstruktionsgröße (siehe 4). Wegen der einfach gehaltenen hier betrachteten Architektur kann diese Anzahl gleichgesetzt werden mit der Ausführungsgeschwindigkeit einer fiktiv-physischen Realisation der Architektur, da halbleiterbasierte Optimierungen nicht vorhanden sind.

Als Sortieraufgabe wird stets die Zufallsvariable

$$d_n \sim \text{Unif}\left(\{1,\ldots,n\} \xrightarrow{\sim} \{1,\ldots,n\}_{W_{32,\pm}}\right)$$

mit Variabler n und mehreren Sortierungsrealisationen betrachtet. Dabei bezeichne $\{1,\ldots,n\}$ die Menge der Datenfeldindizes, $\{1,\ldots,n\}_{W_{32,\pm}}$ die der Datenfeldeinträge, $A \xrightarrow{\sim} B := \{f : A \to B \,|\, f \text{ bijektiv}\}$ und Unif die uniforme Verteilung auf einer endlichen Menge.
Es werden Mittelwert und empirische Standardabweichung einer Monte-Carlo-Simulation eben beschriebener Parameter errechnet und deren Visualisierung im Folgenden besprochen.

3 Sortieralgorithmen

Modelliert man ein Datenfeld der Länge n bestehend aus vorzeichenbehafteten 32-Bit-Zahlen $W_{32,\pm} := \{-2^{31},\ldots,2^{31}-1\} \subset \mathbb{Z}$ als eine Abbildung

$$d : \{1,\ldots,n\} \to W_{32,\pm},$$

so lässt sich die Aufgabe des Sortierens als algorithmisches Auffinden einer Abbildung

$$s : \{1,\ldots,n\} \to \{1,\ldots,n\}$$

formulieren, sodass $d \circ s$ als Abbildung isoton bezüglich der Standardordnung auf $\mathbb{Z}$ ist. Das Datenfeld $d \circ s$ heißt in diesem Fall *sortiert*. Man beachte, dass keine Aussage über die Stabilität des Sortierverfahrens getroffen wird.

3.1 Quicksort

Quicksort ist eine durch Pivotwahl parametrisierte Sortierverfahrensfamilie. Der hier betrachtete Vertreter jener Familie wählt als Pivotelement stets das erste Element der Liste aus. Aufgrund der Uniformität des zu sortierenden Datenfelds entsteht so keine Asymmetrie in der Durchführung des Sortierens, jedoch ist es die am einfachsten und auch am laufzeitschnellsten zu implementierende Variante.[1]
Für Näheres siehe den Wikipedia-Artikel `https://en.wikipedia.org/wiki/Quicksort`[2].
Für den Assembler-Quelltext siehe 5.1. Nach mehreren erfolglosen Versuchen, Quicksort direkt in Joy Assembler zu implementieren, wählte ich den Weg der *menschlichen Kompilation*: Ich implementierte Quicksort in der Hochsprache C samt Testumgebung und formulierte diese Implementation in mehreren Schritten so um, dass eine direkte Übertragung der testbaren C-Implementation in ein Joy-Assembler-Programm möglich war. Für Genaueres zu diesem Prozess siehe 6.

[1] Andere erdenkbare Varianten wären ein Pivotelement uniform verteilt aus dem zu teilenden Datenfeld auszuwählen oder durch eine lineare Betrachtung des gesamten Datenfelds zu versuchen, einen möglichst halbierenden Schnitt mittels des Pivotelementes zu erreichen.

[2] Siehe `https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=982549872`. Abgerufen 2020-10-19.

3.2 Shellsort

Shellsort ist eine durch Elementabstandswahl (englisch *gap*) parametriesierte Familie an Sortieralgorithmen. Hier betrachtet werden die Folgen

- $\left(1\right)$, das heißt Insertionsort mit konstanter Basislast aufgrund des Durchlaufens der hier trivialen Folge;
- $\left(1, 4, 9, 20, \dots\right)$, sowie rechts gestutzte Varianten[3];
- $\left(1, 4, 10, 23, \dots\right)$, sowie rechts gestutzte Varianten[4].

Für Näheres siehe den Wikipedia-Artikel `https://en.wikipedia.org/wiki/Shellsort`[5].
Anders als bei Quicksort gelang es mir, den im oben verlinkten Artikel beschriebenen Algorithmus direkt in Joy Assembler zu implementieren. Der Quelltext dieser Implementation ist in 5.2 abgedruckt.

3.3 Vergleich beider Sortierverfahren und ihrer Varianten

Das kleinste betrachtete Regime ist in Abbildung 1 zu finden: $0 \leq n \leq 16$. Es werden nur Quicksort (`qsort`) sowie drei Shellsort-Varianten (`shellsort_1`, `shellsort_4-1` und `shellsort_10-4-1`) dargestellt, da größere Lücken sowie kleine Abänderungen der Lücken in diesem Regime einen so geringen Unterschied machen, dass die Abbildung sonst an Lesbarkeit verlöre.
Bei sowohl $n = 0$ als auch $n = 1$ sieht man bei allen Verfahren eine konstante Laufzeit ohne Varianz – es gibt schließlich nur ein Datenfeld, welches stets den Basisfall darstellt. Man beachte, dass Quicksort hier am schnellsten ist, das heißt Quicksort am schnellsten den Basisfall erkennt und das Datenfeld für sortiert erklärt (zur geringen Basislast von Quicksort später mehr). Die drei Shellsort-Varianten sind mit drei verschiedenen Anzahlen an Lücken parametrisiert. Für $0 \leq n \leq 8$ laufen sie langsamer, je größer die Anzahl der Lücken ist. Es zeichnet sich die Notwendigkeit ab, jede Lücke anzuschauen (siehe Quelltext 5.2, Zeile 100).
Ab $n > 1$ vermischen sich die Laufzeitkurven: Quicksort schneidet erkennbar schlechter ab, wobei die drei Shellsort-Varianten nahe beisammen noch in der Reihenfolge ihrer Lückenanzahl laufen. Bei ungefähr $n > 8$ beginnt `shellsort_1` erstmals im Mittel langsamer zu laufen als `shellsort_4-1` und `shellsort_10-4-1`. Ab $n \geq 11$ ist es das langsamste Verfahren. Ein Unterschied zwischen den beiden anderen Shellsort-Verfahren ist kaum erkennbar. Die Lücke der Länge 10 birgt einen gewissen konstanten Laufzeitnachteil, *kann* jedoch erst ab $n > 10$ überhaupt einen Vorteil aufweisen; im Regime $0 \leq n \leq 16$ schneidet `shellsort_10-4-1` in der Abbildung erkennbar stets schlechter ab als `shellsort_4-1`.

[3] Siehe `https://oeis.org/A108870`: *Tokuda's good set of increments for Shell sort.* Abgerufen 2020-10-19.

[4] Siehe `https://oeis.org/A102549`: *Optimal (best known) sequence of increments for shell sort algorithm.* Abgerufen 2020-10-19.

[5] Siehe `https://en.wikipedia.org/w/index.php?title=Shellsort&oldid=981554038`. Abgerufen 2020-10-19.

Quicksort hingegen durchlebt alle Phasen des Rennens: als schnellster Start ist Quicksort zwischen $2 \leq n < 11$ das langsamste Verfahren, vermischt sich mit den Kontrahenten während der Datenfeldlängen $11 \leq n \leq 15$ und etabliert sich als marginal schnelleres Verfahren bei $n = 16$. Diesen Leistungsvorsprung wird Quicksort für $n > 16$ nur weiter ausbauen.
Man bemerke, dass die hier betrachte Quicksort-Variante die schnellste erdenkliche Pivotwahl trifft: schlicht das erste Element des zu sortierenden Datenfelds. Wählte man eine aufwendigere Heuristik zur Pivotwahl, so erhöhte sich auch die konstante Basislast, wodurch Shellsort bei kleinen Datenfeldlängen stärker und länger einen Vorsprung erzielen könnte.

Im Regime $0 \leq n \leq 32$ (siehe Abbildung 2) ist deutlich der Leistungsabfall des Insertionsort-Algorithmus `shellsort_1` zu sehen: ab $n \geq 19$ dauert stets das Sortieren des in einer Standardabweichung für den Algorithmus einfachst zu sortierenden Datenfelds länger als das des in einer Standardabweichung schwierigst zu sortierende aller anderen Verfahren. Auch lässt sich der Ausbau des Vorsprungs von Quicksort verglichen mit allen anderen Verfahren erkennen. Wie erwartet separieren sich auch `shellsort_4-1` und `shellsort_10-4-1` ein wenig, wobei die zusätzliche Lücke sich ab $n \geq 25$ als nützlich zu erweisen scheint.

Betrachtet man noch längere Datenfelder (siehe Abbildung 3), so wächst die Kluft zwischen allen betrachteten Verfahren weiter, ihre Performanz entspricht den Erwartungen.

Um Unterschiede zwischen `shellsort_10-4-1` und `shellsort_9-4-1` erkennen zu können, muss das Laufzeitverhalten bei großem n getestet werden (siehe Abbildung 4). Diese Abbildung zeigt weiter die Verfahren `shellsort_20-9-4-1` und `shellsort_23-10-4-1`. Man sieht hier gut, wie die Hinzunahme an Lücken die Laufzeit verbessert, sowie dass die als optimal behaupteten Lücken auch wirklich besser als die anderen betrachteten abschneiden. Der Vorsprung Quicksorts und die Ineffizienz Insertionsorts werden durch diese Abbildung besonders stark verdeutlicht.

In Abbildung 5 wird erneut der Bereich $0 \leq n \leq 1024$ betrachtet, hier jedoch ohne Insertionsort (`shellsort_1`), um die Unterschiede zwischen ähnlichen Lückengrößen besser sehen zu können.

3.4 Abbildungen

3.4.1 Kleines Regime ($0 \leq n \leq 16$)

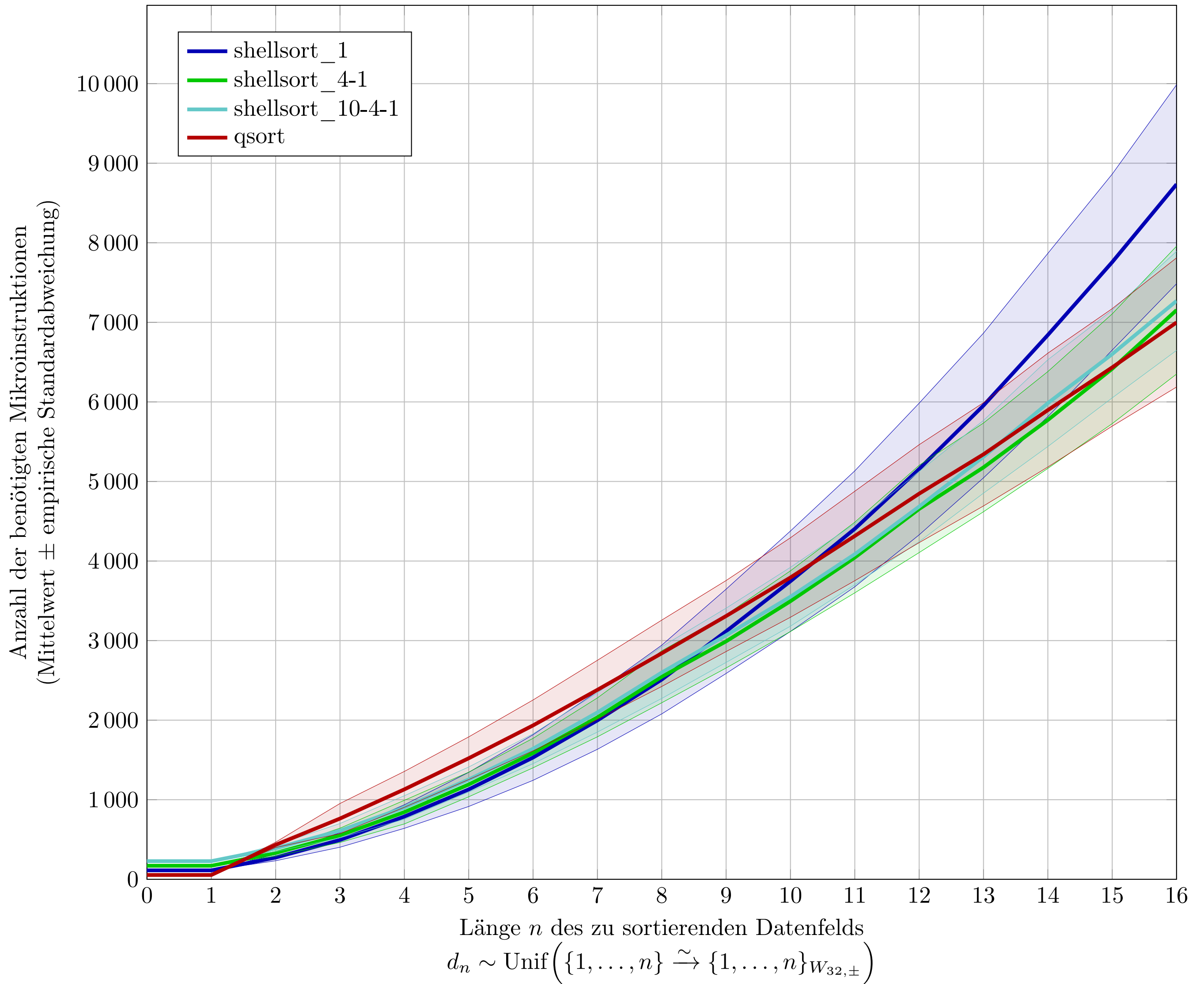

Abbildung 1: Monte-Carlo-Simulation mit 2048 Iterationen der Datenfeldlängen $0 \leq n \leq 16$, gemessen mit Schrittlänge 1 im Zeitraum 2020-11-02T15:04:02Z+01:00 bis 2020-11-02T19:40:25Z+01:00.

3.4.2 Mittleres Regime ($0 \leq n \leq 32$)

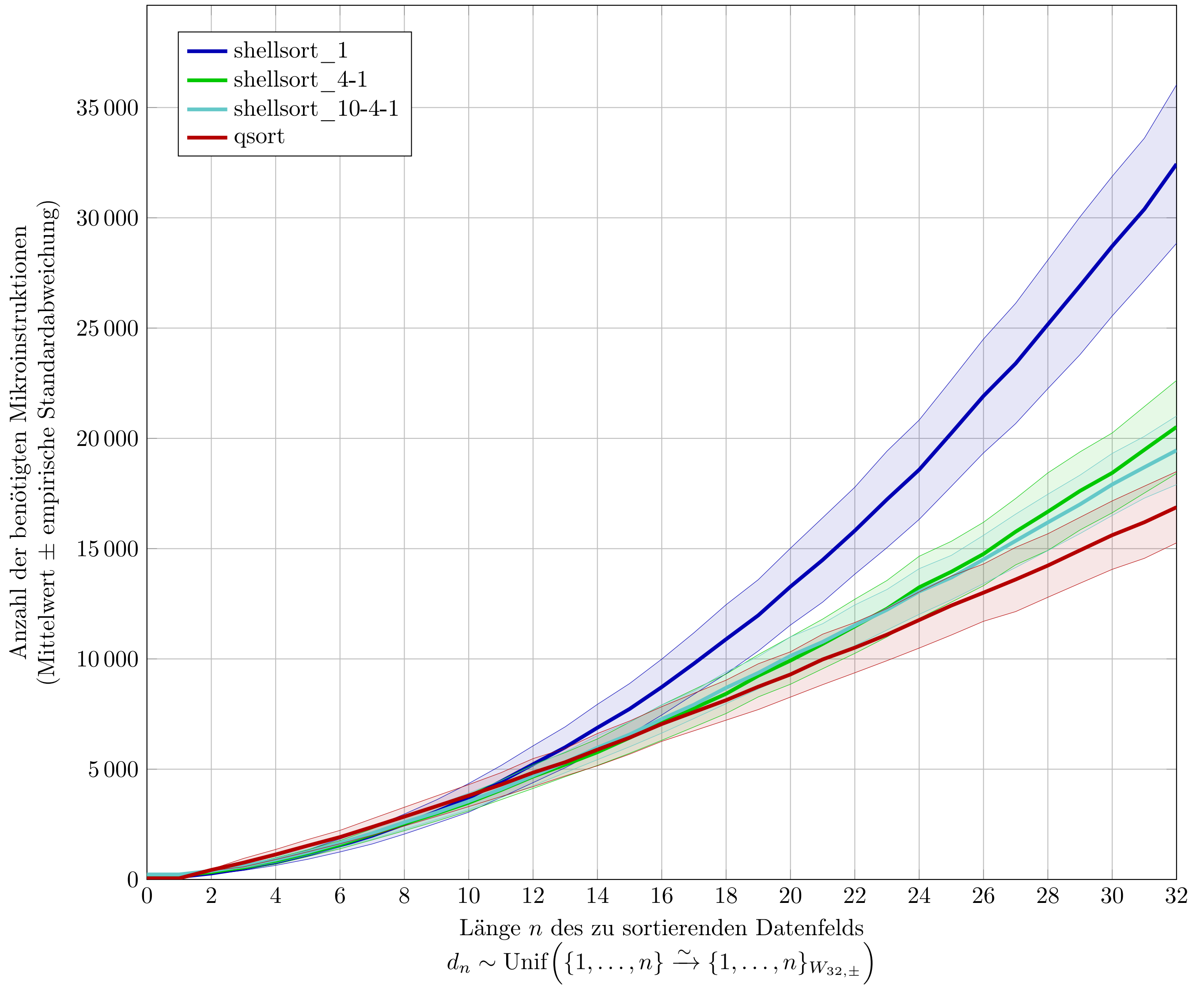

Abbildung 2: Monte-Carlo-Simulation mit 1024 Iterationen der Datenfeldlängen $0 \leq n \leq 32$, gemessen mit Schrittlänge 1 im Zeitraum 2020-11-02T19:40:25Z+01:00 bis 2020-11-03T00:09:00Z+01:00.

3.4.3 Großes Regime ($0 \leq n \leq 64$)

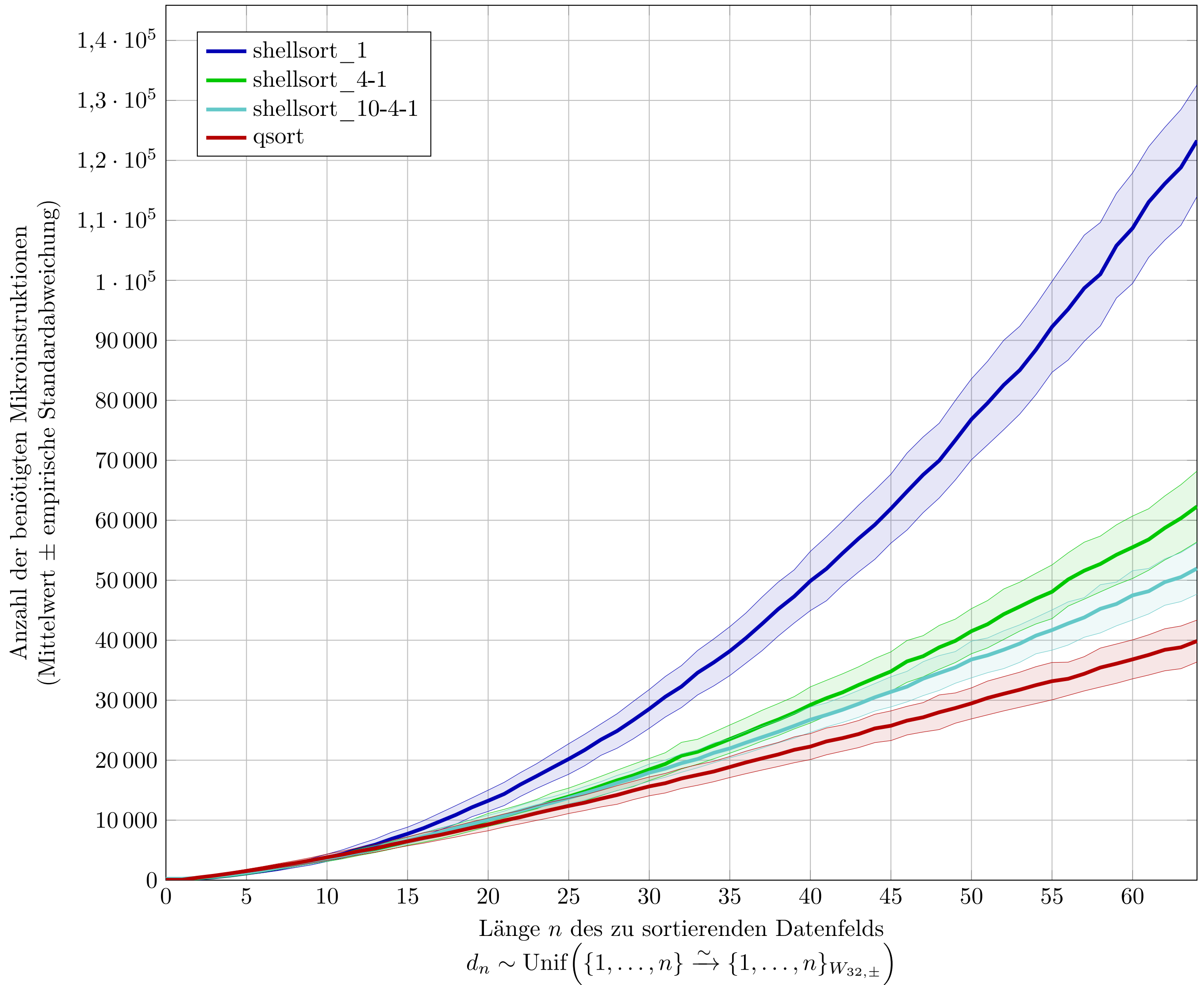

Abbildung 3: Monte-Carlo-Simulation mit 512 Iterationen der Datenfeldlängen $0 \leq n \leq 64$, gemessen mit Schrittlänge 1 im Zeitraum 2020-11-03T00:09:00Z+01:00 bis 2020-11-03T04:33:52Z+01:00.

3.4.4 Riesiges Regime ($0 \leq n \leq 1024$)

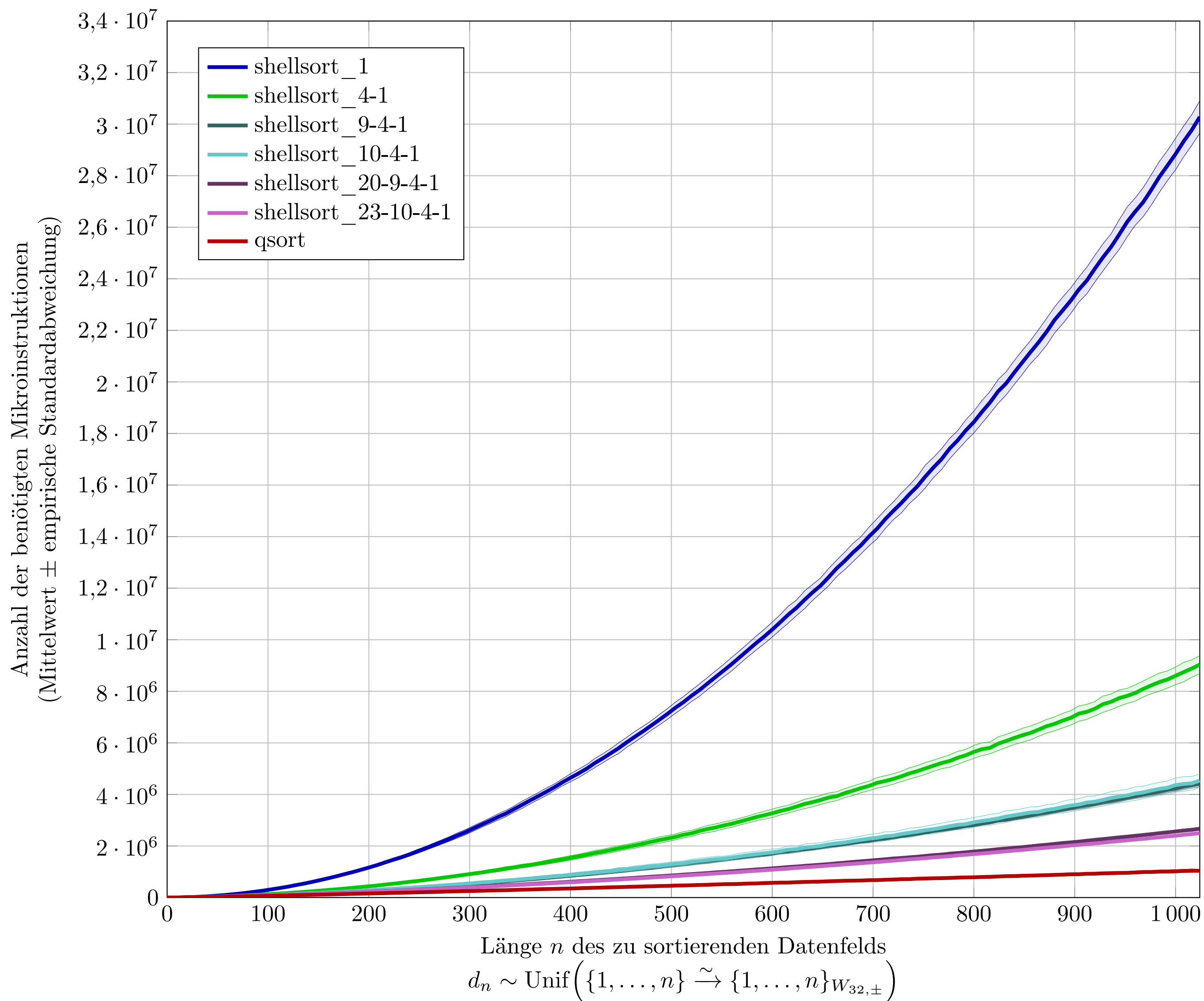

Abbildung 4: Monte-Carlo-Simulation mit 256 Iterationen der Datenfeldlängen $0 \leq n \leq 1024$, gemessen mit Schrittlänge 8 im Zeitraum 2020-11-05T09:47:37Z+01:00 bis 2020-11-05T19:45:31Z+01:00.

3.4.5 Riesiges Regime ohne Insertionsort ($0 \leq n \leq 1024$)

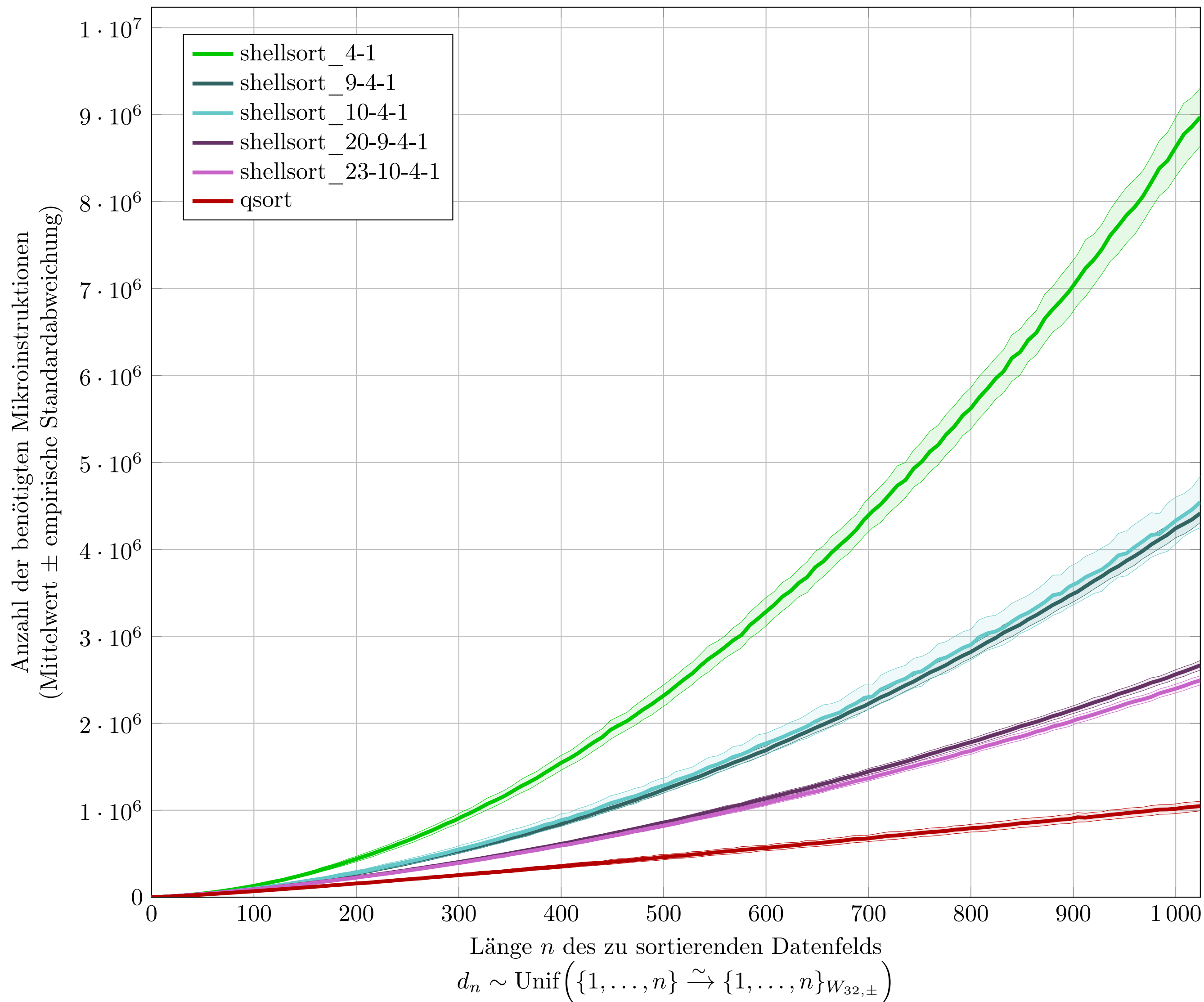

Abbildung 5: Monte-Carlo-Simulation mit 256 Iterationen der Datenfeldlängen $0 \leq n \leq 1024$, gemessen mit Schrittlänge 8 im Zeitraum 2020-11-14T15:23:03Z+01:00 bis 2020-11-14T23:12:30Z+01:00.

4 Mikroinstruktionen

Anfangs nahm ich als Zeitparameter die Anzahl der ausgeführten Instruktionen. Aufgrund der variierenden Komplexität jener insbesondere in Bezug auf den Stapel, war jedoch Quicksort der klare Sieger, selbst bei den kleinsten Datenfeldlängen. Aufgrund dessen schien es methodisch sinnvoll, den verschiedenen Instruktionen verschiedene Ausführungszeiten zuzuordnen. Um diesem Prozess der Willkür nicht Tür und Tor zu öffnen, habe ich versucht, die benötigte Anzahl an Mikroinstruktionen – und damit die Ausführungszeit einer Instruktion – durch eine fiktiv-hardwarenahe Implementation zu bestimmen.
Ich bin mir dessen bewusst, dass andere Wahlen der unten aufgeführten Anzahlen an Mikroinstruktionen zu wesentlich anderen Algorithmengüten führen. Mit der hier getroffene Wahl habe ich mit meinen begrenzten Hardwarekenntnissen versucht, einen fiktiven Schaltkreis nachzuempfinden. Insbsondere gibt es keine besondere Unterstützung von Stapelinstruktionen, wodurch Shellsort im Regime kurzer Datenfelder besser als Quicksort abschneiden kann.

4.1 Mikroinstruktionen (Abschnitt *nop*)

$$\texttt{nop} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow 0 \end{cases}$$

4.2 Mikroinstruktionen (Abschnitt *memory*)

$$\texttt{lda} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{ldb} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{B} \end{cases}$$

$$\texttt{sta} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{Speicher} \end{cases}$$

$$\texttt{stb} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{B} \\ \text{Bus} \rightarrow \text{Speicher} \end{cases}$$

$$\texttt{lia} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \twoheadrightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{sia} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \twoheadrightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{Speicher} \end{cases}$$

$$\texttt{lpc} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{P} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{spc} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{lya} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher}^{1/4} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{sya} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{Speicher}^{1/4} \end{cases}$$

4.3 Mikroinstruktionen (Abschnitt *jumps*)

$$\texttt{jmp} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jn} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jnn} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jz} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jnz} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jp} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jnp} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{je} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{jne} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \overset{?}{\leftarrow} \text{P} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

4.4 Mikroinstruktionen (Abschnitt *stack*)

$$\texttt{cal} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{P} \\ \text{Bus} \rightarrow \text{Speicher} \\ \\ \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{A} \\ 4 \twoheadrightarrow \text{A} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{S} \\ \\ \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{ret} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{A} \\ -4 \twoheadrightarrow \text{A} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{S} \\ \\ \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{P} \end{cases}$$

$$\texttt{psh} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{Speicher} \\ \\ \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{A} \\ 4 \twoheadrightarrow \text{A} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{S} \end{cases}$$

$$\texttt{pop} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{A} \\ -4 \twoheadrightarrow \text{A} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{S} \\ \\ \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{lsa} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \twoheadrightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Speicher} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{ssa} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \twoheadrightarrow \text{Speicheradresse} \\ \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{Speicher} \end{cases}$$

$$\texttt{lsc} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{S} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{ssc} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \text{Bus} \rightarrow \text{S} \end{cases}$$

4.5 Mikroinstruktionen (Abschnitt *reg. A*)

$$\texttt{mov} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{Argument} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{not} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft \neg \end{cases}$$

$$\texttt{shl} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft \cdot 2^{\text{statisches Argument}} \end{cases}$$

$$\texttt{shr} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft \cdot 2^{-\text{statisches Argument}} \end{cases}$$

$$\texttt{inc} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft +\text{statisches Argument} \end{cases}$$

$$\texttt{dec} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft -\text{statisches Argument} \end{cases}$$

$$\texttt{neg} \quad :\Longleftrightarrow \quad \begin{cases} \text{A} \circlearrowleft - \end{cases}$$

4.6 Mikroinstruktionen (Abschnitt *reg.* A, B)

$$\texttt{swp} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \rightleftharpoons \text{A} \\ \text{Bus} \rightarrow \text{B} \end{cases}$$

$$\texttt{and} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \xrightarrow{\wedge} \text{A} \end{cases}$$

$$\texttt{or} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \xrightarrow{\vee} \text{A} \end{cases}$$

$$\texttt{xor} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \xrightarrow{\oplus} \text{A} \end{cases}$$

$$\texttt{add} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \xrightarrow{+} \text{A} \end{cases}$$

$$\texttt{sub} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{B} \\ \text{Bus} \xrightarrow{-} \text{A} \end{cases}$$

4.7 Mikroinstruktionen (Abschnitt *i/o*)

$$\texttt{get} \quad :\Longleftrightarrow \quad \begin{cases} \ldots IO \ldots \\ \text{Bus} \leftarrow \text{IO} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{gtc} \quad :\Longleftrightarrow \quad \begin{cases} \ldots IO \ldots \\ \text{Bus} \leftarrow \text{IO} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

$$\texttt{ptu} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \ldots IO \ldots \\ \text{Bus} \rightarrow \text{IO} \end{cases}$$

$$\texttt{pts} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \ldots IO \ldots \\ \text{Bus} \rightarrow \text{IO} \end{cases}$$

$$\texttt{ptb} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \ldots IO \ldots \\ \text{Bus} \rightarrow \text{IO} \end{cases}$$

$$\texttt{ptc} \quad :\Longleftrightarrow \quad \begin{cases} \text{Bus} \leftarrow \text{A} \\ \ldots IO \ldots \\ \text{Bus} \rightarrow \text{IO} \end{cases}$$

4.8 Mikroinstruktionen (Abschnitt *rnd*)

$$\texttt{rnd} \quad :\Longleftrightarrow \quad \begin{cases} \ldots \mathit{IO} \ldots \\ \text{Bus} \leftarrow \text{IO} \\ \text{Bus} \rightarrow \text{A} \end{cases}$$

4.9 Mikroinstruktionen (Abschnitt *hlt*)

$$\texttt{hlt} \quad :\Longleftrightarrow \quad \begin{cases} 1 \rightarrow \text{H} \end{cases}$$

5 Quelltext

5.1 Quicksort

```
1  qsort:
2      ; load local values from stack
3      lsa -12
4      sta @qsort$p-array-begin
5      lsa -8
6      sta @qsort$p-array-end
7
8      ; base case
9      lda @qsort$p-array-end
10     ldb @qsort$p-array-begin
11     sub
12     dec 4
13     jnp @qsort$return
14
15     ; initialize
16     ldb @qsort$p-array-begin
17     lia
18     sta @qsort$pivot
19     lda @qsort$p-array-begin
20     sta @qsort$p-left
21
22     ; loop
23     lda @qsort$p-array-begin
24     sta @qsort$p
25     qsort$loop:
26         ; loop through array; the first element
27         ; is chosen as a pivot and thus
28         ; immediately skipped on the first iteration
29         lda @qsort$p
30         inc 4
31         sta @qsort$p
32
33         ; array end
34         lda @qsort$p-array-end
35         ldb @qsort$p
36         sub
37         jnp @qsort$pool
38
39         ; should the element be swapped?
40         ldb @qsort$p
41         lia
42         ldb @qsort$pivot
43         sub
44         jnn @qsort$loop
45
46         ; swap element
47         lda @qsort$p-left
48         inc 4
49         sta @qsort$p-left
50         ; swap
```

```
51          ldb @qsort$p-left
52          lia
53          sta @qsort$tmp1
54          ldb @qsort$p
55          lia
56          sta @qsort$tmp2
57          ldb @qsort$p
58          lda @qsort$tmp1
59          sia
60          ldb @qsort$p-left
61          lda @qsort$tmp2
62          sia
63      jmp @qsort$loop
64      qsort$pool:
65
66      ; swap
67      ldb @qsort$p-left
68      lia
69      sta @qsort$tmp1
70      ldb @qsort$p-array-begin
71      lia
72      sta @qsort$tmp2
73      ldb @qsort$p-array-begin
74      lda @qsort$tmp1
75      sia
76      ldb @qsort$p-left
77      lda @qsort$tmp2
78      sia
79
80
81      ; recur
82          lda @qsort$p-left
83          inc 4
84          psh
85          lda @qsort$p-array-end
86          psh
87
88          lda @qsort$p-array-begin
89          psh
90          lda @qsort$p-left
91          psh
92
93          cal @qsort
94          pop
95          pop
96          ; now the pseudo-local global
97          ; variables are cluttered by the call
98
99          ; pivot is already sorted
100
101         cal @qsort
102         pop
103         pop
104
```

```
105     qsort$return:
106         ret
107
108     qsort$p-array-begin:
109         data [1]
110     qsort$p-array-end:
111         data [1]
112     qsort$pivot:
113         data [1]
114     qsort$p-left:
115         data [1]
116     qsort$p:
117         data [1]
118     qsort$tmp1:
119         data [1]
120     qsort$tmp2:
121         data [1]
```

5.2 Shellsort

```
 1  shellsort:
 2      ; load local values from stack
 3      lsa -12
 4      sta @shellsort$p-array-begin
 5      lsa -8
 6      sta @shellsort$p-array-end
 7
 8      ; leech lda @shellsort$p-array-end
 9      ldb @shellsort$p-array-begin
10      sub
11      sta @shellsort$n
12
13      mov @gaps
14      sta @shellsort$p-gaps
15
16      shellsort$gap-loop:
17          ldb @shellsort$p-gaps
18          lia
19          sta @shellsort$gap
20
21          ; leech lda @shellsort$gap
22          sta @shellsort$i
23
24          shellsort$i-loop:
25              lda @shellsort$i
26                  sta @shellsort$j
27              ldb @shellsort$n
28              sub
29              jnn @shellsort$i-pool
30
31              ldb @shellsort$p-array-begin
32              lda @shellsort$i
33              add
34              swp
```

```
35              lia
36              sta @shellsort$tmp
37
38              ; previously sta @shellsort$j
39              shellsort$j-loop:
40                  ;     !(j >= gap && a[j - gap] > tmp)
41                  ; <=> j < gap || a[j - gap] <= tmp
42                  ; <=> j - gap < 0 || a[j - gap] - tmp <= 0
43                  lda @shellsort$j
44                  ldb @shellsort$gap
45                  sub
46                  jn @shellsort$j-pool
47                  ;;;
48                  lda @shellsort$j
49                  ldb @shellsort$gap
50                  sub
51                  ldb @shellsort$p-array-begin
52                  add
53                  swp
54                  lia
55                  ldb @shellsort$tmp
56                  sub
57                  jnp @shellsort$j-pool
58
59                  lda @shellsort$j
60                  ldb @shellsort$gap
61                  sub
62                  ldb @shellsort$p-array-begin
63                  add
64                  swp
65                  lia
66                  sta @shellsort$tmp2
67                  ;;;
68                  lda @shellsort$j
69                  ldb @shellsort$p-array-begin
70                  add
71                  swp
72                  lda @shellsort$tmp2
73                  sia
74
75                  lda @shellsort$j
76                  ldb @shellsort$gap
77                  sub
78                  sta @shellsort$j
79              jmp @shellsort$j-loop
80              shellsort$j-pool:
81
82              ldb @shellsort$p-array-begin
83              lda @shellsort$j
84              add
85              swp
86              lda @shellsort$tmp
87              sia
88
```

```
89              lda @shellsort$i
90              inc 4
91              sta @shellsort$i
92          jmp @shellsort$i-loop
93          shellsort$i-pool:
94
95          lda @shellsort$p-gaps
96          inc 4
97          sta @shellsort$p-gaps
98          swp
99          lia
100         jz @shellsort$gap-pool
101     jmp @shellsort$gap-loop
102     shellsort$gap-pool:
103         ret
104
105     shellsort$p-array-begin:
106         data [1]
107     shellsort$p-array-end:
108         data [1]
109     shellsort$n:
110         data [1]
111     shellsort$p-gaps:
112         data [1]
113     shellsort$gap:
114         data [1]
115     shellsort$i:
116         data [1]
117     shellsort$tmp:
118         data [1]
119     shellsort$j:
120         data [1]
121     shellsort$tmp2:
122         data [1]
```

6 Menschliche Kompilation

Um Algorithmen auf einer fiktiven Architektur empirisch zu betrachten, muss man sowohl die Architektur implementieren als auch in ihr die zu betrachtenden Algorithmen. Bei einem nicht-rekursiven Algorithmus wie Shellsort ist eine sehr direkte Übertragung eines imperativ geschriebenen Algorithmus möglich. Macht der Algorithmus jedoch von nicht-endständiger Rekursion Gebrauch, so bedarf es besonderer Detailverliebtheit, die Stapelrahmen händisch auszulegen, aufzubauen und in richtiger Reihenfolge wieder abzubauen.
Im Folgenden sind verschiedene Stufen meines Prozesses dargestellt, um eine von mir erdachte Quicksort-Implementation in C in mehreren Schritten auf die Joy-Assembler-Architektur umzuschreiben. Semantische Quelltexteinheiten sind gleichfarbig markiert, sich von einer Stufe auf die andere veränderte Zeilen sind kursiv gesetzt.

Anzumerken ist, dass ich `typedef signed long long int word_t;` anstatt von etwa `typedef int32_t word_t;` schreibe. Diese Typdefinition habe ich gewählt, damit für einen Zeiger `word_t *p` die doppelte Typumwandlung `(word_t *) ((word_t) p)` verlustfrei durchgeführt werden kann.
Auf Architekturen bei denen `sizeof (void *) > sizeof (signed long long int)` gilt müsste man diese Typedefinition anpassen.

6.1 C-Konstrukte auflösen

Implizite `return`-Ausdrücke werden explizit formuliert und mit einem Sprung an das Routinenende ersetzt. Die `for`-Schleife wird mittels `goto` dargestellt.

Quelltext „kompilationsschritte/qsort0.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort0(
007     word_t *p_array_begin, word_t *p_array_end
008 ) {
009
010
011     if (p_array_end - p_array_begin <= 1)
012         return;
013
014     word_t pivot = *p_array_begin;
015     word_t *p_left = p_array_begin;
016
017     for (
018         word_t *p = p_array_begin + 1;
019         p < p_array_end;
020         ++p
021     ) {
022         if (*p >= pivot)
023             continue;
024         ++p_left;
025
026         swp_ptr(p_left, p);
027
```

Quelltext „kompilationsschritte/qsort1.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort1(
007     word_t *p_array_begin, word_t *p_array_end
008 ) {
009
010
011     if (p_array_end - p_array_begin <= 1)
012         goto qsort1_return;
013
014     word_t pivot = *p_array_begin;
015     word_t *p_left = p_array_begin;
016
017     word_t *p = p_array_begin;
018     qsort1_loop:
019         ++p;
020         if (!(p < p_array_end))
021             goto qsort1_pool;
022         if (*p >= pivot)
023             goto qsort1_loop;
024         ++p_left;
025
026         swp_ptr(p_left, p);
027
```

Quelltext „kompilationsschritte/qsort0.c“

```
028     }
029

030     swp_ptr(p_array_begin, p_left);
031
032     qsort0(p_array_begin, p_left);
033     /* pivot is already sorted */
034     qsort0(p_left+1, p_array_end);
035
036 }
```

Quelltext „kompilationsschritte/qsort1.c“

```
028     goto qsort1_loop;
029     qsort1_pool:
030
031     swp_ptr(p_array_begin, p_left);
032
033     qsort1(p_array_begin, p_left);
034     /* pivot is already sorted */
035     qsort1(p_left+1, p_array_end);
036
037     qsort1_return:
038         return;
039 }
```

6.2 Typsemantik auflösen

Jede Variable wird im K&R-Stil anfangs deklariert und bekommt den Typ `word_t`; falls C die Zeigersemantik benötigt, wird eine explizite Typumwandlung angegeben. Inkrementoperatoren werden aufgelöst, implizite Zeigerarithmetikinformation mit `sizeof (word_t)` ergänzt.

Quelltext „kompilationsschritte/qsort1.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort1(
007     word_t *p_array_begin, word_t *p_array_end
008 ) {
009










010


011     if (p_array_end - p_array_begin <= 1)
012         goto qsort1_return;
013
014     word_t pivot = *p_array_begin;
015     word_t *p_left = p_array_begin;
016
017     word_t *p = p_array_begin;
018     qsort1_loop:
019         ++p;
020         if (!(p < p_array_end))
021             goto qsort1_pool;
022         if (*p >= pivot)
023             goto qsort1_loop;
024         ++p_left;
025
026         swp_ptr(p_left, p);
027
```

Quelltext „kompilationsschritte/qsort2.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort2(
007     word_t *_p_array_begin, word_t *_p_array_end
008 ) {
009     word_t a, b;
010
011     word_t p_array_begin = 0;
012     word_t p_array_end = 0;
013     word_t pivot = 0;
014     word_t p_left = 0;
015     word_t p = 0;
016
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
020     if (p_array_end - p_array_begin <= 1)
021         goto qsort2_return;
022
023     pivot = *((word_t *) p_array_begin);
024     p_left = (word_t) p_array_begin;
025
026     p = p_array_begin;
027     qsort2_loop:
028         p = p + sizeof (word_t);
029         if (!(p < p_array_end))
030             goto qsort2_pool;
031         if (*((word_t *) p) >= pivot)
032             goto qsort2_loop;
033         p_left = p_left + sizeof (word_t);
034
035         swp_ptr(
036             (word_t *) p_left,
037             (word_t *) p);
```

Quelltext „kompilationsschritte/qsort1.c“

```
028     goto qsort1_loop;
029     qsort1_pool:
030
031     swp_ptr(p_array_begin, p_left);
032
033     qsort1(p_array_begin, p_left);
034     /* pivot is already sorted */
035     qsort1(p_left+1, p_array_end);
036
037     qsort1_return:
038         return;
039 }
```

Quelltext „kompilationsschritte/qsort2.c“

```
038
039     goto qsort2_loop;
040     qsort2_pool:
041
042     swp_ptr(
043         (word_t *) p_array_begin,
044         (word_t *) p_left);
045
046     qsort2(
047         (word_t *) p_array_begin,
048         (word_t *) p_left);
049     /* pivot is already sorted */
050     qsort2(
051         (word_t *) (p_left
052             + sizeof (word_t)),
053         (word_t *) p_array_end);
054
055     qsort2_return:
056         return;
057 }
```

6.3 if-Ausdrücke auflösen

Alle `if`-Ausdrücke werden mittels `goto` dargestellt bis auf bedingte Sprünge der Form `if (...) goto ...;`.

Quelltext „kompilationsschritte/qsort2.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort2(
007     word_t *_p_array_begin, word_t *_p_array_end
008 ) {
009     word_t a, b;
010
011     word_t p_array_begin = 0;
012     word_t p_array_end = 0;
013     word_t pivot = 0;
014     word_t p_left = 0;
015     word_t p = 0;
016
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
020     if (p_array_end - p_array_begin <= 1)
021         goto qsort2_return;
022
023     pivot = *((word_t *) p_array_begin);
024     p_left = (word_t) p_array_begin;
025
026     p = p_array_begin;
027     qsort2_loop:
028         p = p + sizeof (word_t);
029         if (!(p < p_array_end))
```

Quelltext „kompilationsschritte/qsort3.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort3(
007     word_t *_p_array_begin, word_t *_p_array_end
008 ) {
009     word_t a, b;
010
011     word_t p_array_begin = 0;
012     word_t p_array_end = 0;
013     word_t pivot = 0;
014     word_t p_left = 0;
015     word_t p = 0;
016
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
020     a = p_array_end;
021     b = p_array_begin;
022     a -= b;
023     a -= sizeof (word_t);
024     if (a <= 0) goto qsort3_return;
025
026     pivot = *((word_t *) p_array_begin);
027     p_left = p_array_begin;
028
029     p = p_array_begin;
030     qsort3_loop:
031         p = p + sizeof (word_t);
032         a = p_array_end;
```

Quelltext „kompilationsschritte/qsort2.c“

```
030            goto qsort2_pool;
031        if (*((word_t *) p) >= pivot)
032            goto qsort2_loop;
033        p_left = p_left + sizeof (word_t);
034
035        swp_ptr(
036            (word_t *) p_left,
037            (word_t *) p);
038
039    goto qsort2_loop;
040    qsort2_pool:
041
042    swp_ptr(
043        (word_t *) p_array_begin,
044        (word_t *) p_left);
045
046    qsort2(
047        (word_t *) p_array_begin,
048        (word_t *) p_left);
049    /* pivot is already sorted */
050    qsort2(
051        (word_t *) (p_left
052            + sizeof (word_t)),
053        (word_t *) p_array_end);
054
055    qsort2_return:
056        return;
057 }
```

Quelltext „kompilationsschritte/qsort3.c“

```
033        b = p;
034        a -= b;
035        if (a <= 0) goto qsort3_pool;
036
037        a = *((word_t *) p);
038        b = pivot;
039        a -= b;
040        if (a >= 0) goto qsort3_loop;
041        p_left = p_left + sizeof (word_t);
042
043        swp_ptr(
044            (word_t *) p_left,
045            (word_t *) p);
046
047    goto qsort3_loop;
048    qsort3_pool:
049
050    swp_ptr(
051        (word_t *) p_array_begin,
052        (word_t *) p_left);
053
054    qsort3(
055        (word_t *) p_array_begin,
056        (word_t *) p_left);
057    /* pivot is already sorted */
058    qsort3(
059        (word_t *) (p_left
060            + sizeof (word_t)),
061        (word_t *) p_array_end);
062
063    qsort3_return:
064        return;
065 }
```

6.4 Ausdrücke linearisieren

Die Routine `swp_ptr` wird durch ihren Rumpf ersetzt und verschachtelte Ausdrücke werden mittels der Variablen `a` und `b` linearisiert, welche die Joy-Assembler-Register A und B repräsentieren.
Da der Quelltext erheblich an Länge gewinnt, werden Kommentare für alle großen Algorithmusabschnitte eingefügt.

Quelltext „kompilationsschritte/qsort3.c“

```
001 typedef signed long long int word_t;
002
003 void swp_ptr(word_t *p, word_t *q) {
004     word_t t = *p; *p = *q; *q = t; }
005
006 void qsort3(
007     word_t *_p_array_begin, word_t *_p_array_end
008 ) {
009     word_t a, b;
010
011     word_t p_array_begin = 0;
012     word_t p_array_end = 0;
013     word_t pivot = 0;
014     word_t p_left = 0;
015     word_t p = 0;
```

Quelltext „kompilationsschritte/qsort4.c“

```
001 typedef signed long long int word_t;
002
003 void qsort4(
004     word_t *_p_array_begin, word_t *_p_array_end
005 ) {
006     word_t a, b;
007
008     word_t p_array_begin = 0;
009     word_t p_array_end = 0;
010     word_t pivot = 0;
011     word_t p_left = 0;
012     word_t p = 0;
```

Quelltext „kompilationsschritte/qsort3.c“

```
016
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
020     a = p_array_end;
021     b = p_array_begin;
022     a -= b;
023     a -= sizeof (word_t);
024     if (a <= 0) goto qsort3_return;
025
026     pivot = *((word_t *) p_array_begin);
027     p_left = p_array_begin;
028
029     p = p_array_begin;
030     qsort3_loop:
031         p = p + sizeof (word_t);
032         a = p_array_end;
033         b = p;
034         a -= b;
035         if (a <= 0) goto qsort3_pool;
036
037         a = *((word_t *) p);
038         b = pivot;
039         a -= b;
040         if (a >= 0) goto qsort3_loop;
041         p_left = p_left + sizeof (word_t);
042
043         swp_ptr(
044             (word_t *) p_left,
045             (word_t *) p);
046
047     goto qsort3_loop;
048     qsort3_pool:
049
050     swp_ptr(
051         (word_t *) p_array_begin,
052         (word_t *) p_left);
053
054     qsort3(
```

Quelltext „kompilationsschritte/qsort4.c“

```
013     word_t tmp1 = 0;
014     word_t tmp2 = 0;
015
016     /* load local values from stack */
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
020     /* base case */
021     a = p_array_end;
022     b = p_array_begin;
023     a -= b;
024     a -= sizeof (word_t);
025     if (a <= 0) goto qsort4_return;
026
027     /* initialize */
028     pivot = *((word_t *) p_array_begin);
029     p_left = p_array_begin;
030
031     /* loop */
032     p = p_array_begin;
033     qsort4_loop:
034         /* loop through array; the first element
035            is chosen as a pivot and thus
036            immediately skipped on the first
037            iteration */
038         a = p;
039         a += sizeof (word_t);
040         p = a;
041
042         /* array end */
043         a = p_array_end;
044         b = p;
045         a -= b;
046         if (a <= 0) goto qsort4_pool;
047
048         /* should the element be swapped? */
049         b = p;
050         a = *((word_t *) b);
051         b = pivot;
052         a -= b;
053         if (a >= 0) goto qsort4_loop;
054
055         /* advance left pointer */
056         a = p_left;
057         a += sizeof (word_t);
058         p_left = a;
059
060         /* swap */
061         tmp1 = *((word_t *) p_left);
062         tmp2 = *((word_t *) p);
063         *((word_t *) p_left) = tmp2;
064         *((word_t *) p) = tmp1;
065
066     goto qsort4_loop;
067     qsort4_pool:
068
069     /* swap */
070     tmp1 = *((word_t *) p_array_begin);
071     tmp2 = *((word_t *) p_left);
072     *((word_t *) p_array_begin) = tmp2;
073     *((word_t *) p_left) = tmp1;
```

Quelltext „kompilationsschritte/qsort3.c“

```
055         (word_t *) p_array_begin,
056         (word_t *) p_left);
057     /* pivot is already sorted */
058     qsort3(
059         (word_t *) (p_left
060             + sizeof (word_t)),
061         (word_t *) p_array_end);
062
063     qsort3_return:
064         return;
065 }
```

Quelltext „kompilationsschritte/qsort4.c“

```
074
075     /* recur */
076     {
077         qsort4(
078             (word_t *) p_array_begin,
079             (word_t *) p_left);
080         /* pivot is already sorted */
081         qsort4(
082             (word_t *) (p_left
083                 + sizeof (word_t)),
084             (word_t *) p_array_end);
085     }
086
087     qsort4_return:
088         return;
089 }
```

6.5 Präprozessormakros für Joy-Assembler-Instruktionen

Zuvor vereinfachte Ausdrücke werden mittels Präprozessormakros in Instruktionsform gegossen. Alle lokalen `qsort`-Variablen werden ausschließlich als Speicheradressen referenziert.

Quelltext „kompilationsschritte/qsort4.c“

```
001 typedef signed long long int word_t;
002
003 void qsort4(
004     word_t *_p_array_begin, word_t *_p_array_end
005 ) {
006     word_t a, b;
007
008     word_t p_array_begin = 0;
009     word_t p_array_end = 0;
010     word_t pivot = 0;
011     word_t p_left = 0;
012     word_t p = 0;
013     word_t tmp1 = 0;
014     word_t tmp2 = 0;
015
016     /* load local values from stack */
017     p_array_begin = (word_t) _p_array_begin;
018     p_array_end = (word_t) _p_array_end;
019
```

Quelltext „kompilationsschritte/qsort5.c“

```
001 typedef signed long long int word_t;
002
003 #define SUB a -= b;
004 #define DEC(val) a -= val;
005 #define INC(val) a += val;
006 #define JNP(lbl) if (a <= 0) goto lbl;
007 #define JNN(lbl) if (a >= 0) goto lbl;
008 #define LIA a = *((word_t *) b);
009 #define SIA *((word_t *) b) = a;
010 #define LDA(ptr) a = *((word_t *) (ptr));
011 #define LDB(ptr) b = *((word_t *) (ptr));
012 #define STA(ptr) *((word_t *) (ptr)) = a;
013 #define LSA(o) a = stack[(o) / sizeof (word_t)];
014
015 #define V(pseudoPtr) ((word_t *) (pseudoPtr))
016
017 void qsort5(
018     word_t *_p_array_begin, word_t *_p_array_end
019 ) {
020     word_t a, b;
021
022     word_t *RAM = (word_t[7]) { 0, };
023     word_t ATp_array_begin = (word_t) RAM++,
024            ATp_array_end   = (word_t) RAM++,
025            ATpivot         = (word_t) RAM++,
026            ATp_left        = (word_t) RAM++,
027            ATp             = (word_t) RAM++,
028            ATtmp1          = (word_t) RAM++,
029            ATtmp2          = (word_t) RAM++;
030
031     /* load local values from stack */
032     a = (word_t) _p_array_begin;
033     STA(ATp_array_begin)
034     a = (word_t) _p_array_end;
035     STA(ATp_array_end)
```

Quelltext „kompilationsschritte/qsort4.c“

```
020     /* base case */
021     a = p_array_end;
022     b = p_array_begin;
023     a -= b;
024     a -= sizeof (word_t);
025     if (a <= 0) goto qsort4_return;
026
027     /* initialize */
028     pivot = *((word_t *) p_array_begin);
029     p_left = p_array_begin;
030
031     /* loop */
032     p = p_array_begin;
033     qsort4_loop:
034         /* loop through array; the first element
035            is chosen as a pivot and thus
036            immediately skipped on the first
037            iteration */
038         a = p;
039         a += sizeof (word_t);
040         p = a;
041
042         /* array end */
043         a = p_array_end;
044         b = p;
045         a -= b;
046         if (a <= 0) goto qsort4_pool;
047
048         /* should the element be swapped? */
049         b = p;
050         a = *((word_t *) b);
051         b = pivot;
052         a -= b;
053         if (a >= 0) goto qsort4_loop;
054
055         /* advance left pointer */
056         a = p_left;
057         a += sizeof (word_t);
058         p_left = a;
059
060         /* swap */
061         tmp1 = *((word_t *) p_left);
062         tmp2 = *((word_t *) p);
063         *((word_t *) p_left) = tmp2;
064         *((word_t *) p) = tmp1;
065
066     goto qsort4_loop;
067     qsort4_pool:
```

Quelltext „kompilationsschritte/qsort5.c“

```
036
037     /* base case */
038     LDA(ATp_array_end)
039     LDB(ATp_array_begin)
040     SUB
041     DEC(sizeof (word_t))
042     JNP(qsort5_return)
043
044     /* initialize */
045     LDB(ATp_array_begin)
046     LIA
047     STA(ATpivot)
048     LDA(ATp_array_begin)
049     STA(ATp_left)
050
051     /* loop */
052     LDA(ATp_array_begin)
053     STA(ATp);
054     qsort5_loop:
055         /* loop through array; the first element
056            is chosen as a pivot and thus
057            immediately skipped on the first
058            iteration */
059         LDA(ATp)
060         INC(sizeof (word_t));
061         STA(ATp)
062
063         /* array end */
064         LDA(ATp_array_end)
065         LDB(ATp)
066         SUB
067         JNP(qsort5_pool)
068
069         /* should the element be swapped? */
070         LDB(ATp)
071         LIA
072         LDB(ATpivot)
073         SUB
074         JNN(qsort5_loop)
075
076         /* advance left pointer */
077         LDA(ATp_left)
078         INC(sizeof (word_t))
079         STA(ATp_left)
080
081         /* swap */
082         LDB(ATp_left)
083         LIA
084         STA(ATtmp1)
085         LDB(ATp)
086         LIA
087         STA(ATtmp2)
088         LDB(ATp)
089         LDA(ATtmp1)
090         SIA
091         LDB(ATp_left)
092         LDA(ATtmp2)
093         SIA
094
095     goto qsort5_loop;
096     qsort5_pool:
```

Quelltext „kompilationsschritte/qsort4.c“

```
068
069     /* swap */
070     tmp1 = *((word_t *) p_array_begin);
071     tmp2 = *((word_t *) p_left);
072     *((word_t *) p_array_begin) = tmp2;
073     *((word_t *) p_left) = tmp1;
074
075     /* recur */
076     {
077         qsort4(
078             (word_t *) p_array_begin,
079             (word_t *) p_left);
080         /* pivot is already sorted */
081         qsort4(
082             (word_t *) (p_left
083                 + sizeof (word_t)),
084             (word_t *) p_array_end);
085     }
086
087     qsort4_return:
088         return;
089 }
```

Quelltext „kompilationsschritte/qsort5.c“

```
097
098     /* swap */
099     LDB(ATp_left)
100     LIA
101     STA(ATtmp1)
102     LDB(ATp_array_begin)
103     LIA
104     STA(ATtmp2)
105     LDB(ATp_array_begin)
106     LDA(ATtmp1)
107     SIA
108     LDB(ATp_left)
109     LDA(ATtmp2)
110     SIA
111
112     /* recur */
113     {
114         qsort5(
115             (word_t *) (*V(ATp_array_begin)),
116             (word_t *) (*V(ATp_left)));
117         /* pivot is already sorted */
118         qsort5(
119             (word_t *) ((*V(ATp_left))
120                 + sizeof (word_t)),
121             (word_t *) (*V(ATp_array_end)));
122     }
123
124     qsort5_return:
125         return;
126 }
```

6.6 Stapelemulierung

Mit einem `word_t *stack` wird der Stapel emuliert, welcher von `LSA` gelesen wird. Es wird in einem Kommentar angemerkt, dass der Stapelrahmen wiederaufzunehmen ist.

Quelltext „kompilationsschritte/qsort5.c“

```
001 typedef signed long long int word_t;
002
003 #define SUB a -= b;
004 #define DEC(val) a -= val;
005 #define INC(val) a += val;
006 #define JNP(lbl) if (a <= 0) goto lbl;
007 #define JNN(lbl) if (a >= 0) goto lbl;
008 #define LIA a = *((word_t *) b);
009 #define SIA *((word_t *) b) = a;
010 #define LDA(ptr) a = *((word_t *) (ptr));
011 #define LDB(ptr) b = *((word_t *) (ptr));
012 #define STA(ptr) *((word_t *) (ptr)) = a;
013 #define LSA(o) a = stack[(o) / sizeof (word_t)];
014
015 #define V(pseudoPtr) ((word_t *) (pseudoPtr))
016
017 void qsort5(
018     word_t *_p_array_begin, word_t *_p_array_end
019 ) {
020     word_t a, b;
021
```

Quelltext „kompilationsschritte/qsort6.c“

```
001 typedef signed long long int word_t;
002
003 #define SUB a -= b;
004 #define DEC(val) a -= val;
005 #define INC(val) a += val;
006 #define JNP(lbl) if (a <= 0) goto lbl;
007 #define JNN(lbl) if (a >= 0) goto lbl;
008 #define LIA a = *((word_t *) b);
009 #define SIA *((word_t *) b) = a;
010 #define LDA(ptr) a = *((word_t *) (ptr));
011 #define LDB(ptr) b = *((word_t *) (ptr));
012 #define STA(ptr) *((word_t *) (ptr)) = a;
013 #define LSA(o) a = stack[(o) / sizeof (word_t)];
014
015 #define V(pseudoPtr) ((word_t *) (pseudoPtr))
016
017 void qsort6(
018     word_t *p_array_begin, word_t *p_array_end
019 ) {
020     word_t a, b;
021
```

Quelltext „kompilationsschritte/qsort5.c“

```
022     word_t *RAM = (word_t[7]) { 0, };
023     word_t ATp_array_begin = (word_t) RAM++,
024            ATp_array_end   = (word_t) RAM++,
025            ATpivot         = (word_t) RAM++,
026            ATp_left        = (word_t) RAM++,
027            ATp             = (word_t) RAM++,
028            ATtmp1          = (word_t) RAM++,
029            ATtmp2          = (word_t) RAM++;
030

031     /* load local values from stack */
032     a = (word_t) _p_array_begin;
033     STA(ATp_array_begin)
034     a = (word_t) _p_array_end;
035     STA(ATp_array_end)
036
037     /* base case */
038     LDA(ATp_array_end)
039     LDB(ATp_array_begin)
040     SUB
041     DEC(sizeof (word_t))
042     JNP(qsort5_return)
043
044     /* initialize */
045     LDB(ATp_array_begin)
046     LIA
047     STA(ATpivot)
048     LDA(ATp_array_begin)
049     STA(ATp_left)
050
051     /* loop */
052     LDA(ATp_array_begin)
053     STA(ATp);
054     qsort5_loop:
055         /* loop through array; the first element
056            is chosen as a pivot and thus
057            immediately skipped on the first
058            iteration */
059         LDA(ATp)
060         INC(sizeof (word_t));
061         STA(ATp)
062
063         /* array end */
064         LDA(ATp_array_end)
065         LDB(ATp)
066         SUB
067         JNP(qsort5_pool)
068
069         /* should the element be swapped? */
070         LDB(ATp)
071         LIA
072         LDB(ATpivot)
073         SUB
074         JNN(qsort5_loop)
075
076         /* advance left pointer */
```

Quelltext „kompilationsschritte/qsort6.c“

```
022     word_t *RAM = (word_t[7]) { 0, };
023     word_t ATp_array_begin = (word_t) RAM++,
024            ATp_array_end   = (word_t) RAM++,
025            ATpivot         = (word_t) RAM++,
026            ATp_left        = (word_t) RAM++,
027            ATp             = (word_t) RAM++,
028            ATtmp1          = (word_t) RAM++,
029            ATtmp2          = (word_t) RAM++;
030
031     word_t *stack = 4 + (word_t[4]) {
032         (word_t) 0x00add100,
033         (word_t) p_array_begin,
034         (word_t) p_array_end,
035         (word_t) 0x00000000 };
036
037     /* load local values from stack */
038     LSA(-3 * sizeof (word_t))
039     STA(ATp_array_begin)
040     LSA(-2 * sizeof (word_t))
041     STA(ATp_array_end)
042
043     /* base case */
044     LDA(ATp_array_end)
045     LDB(ATp_array_begin)
046     SUB
047     DEC(sizeof (word_t))
048     JNP(qsort6_return)
049
050     /* initialize */
051     LDB(ATp_array_begin)
052     LIA
053     STA(ATpivot)
054     LDA(ATp_array_begin)
055     STA(ATp_left)
056
057     /* loop */
058     LDA(ATp_array_begin)
059     STA(ATp);
060     qsort6_loop:
061         /* loop through array; the first element
062            is chosen as a pivot and thus
063            immediately skipped on the first
064            iteration */
065         LDA(ATp)
066         INC(sizeof (word_t));
067         STA(ATp)
068
069         /* array end */
070         LDA(ATp_array_end)
071         LDB(ATp)
072         SUB
073         JNP(qsort6_pool)
074
075         /* should the element be swapped? */
076         LDB(ATp)
077         LIA
078         LDB(ATpivot)
079         SUB
080         JNN(qsort6_loop)
081
082         /* advance left pointer */
```

Quelltext „kompilationsschritte/qsort5.c“

```
077         LDA(ATp_left)
078         INC(sizeof (word_t))
079         STA(ATp_left)
080
081         /* swap */
082         LDB(ATp_left)
083         LIA
084         STA(ATtmp1)
085         LDB(ATp)
086         LIA
087         STA(ATtmp2)
088         LDB(ATp)
089         LDA(ATtmp1)
090         SIA
091         LDB(ATp_left)
092         LDA(ATtmp2)
093         SIA
094
095     goto qsort5_loop;
096     qsort5_pool:
097
098     /* swap */
099     LDB(ATp_left)
100     LIA
101     STA(ATtmp1)
102     LDB(ATp_array_begin)
103     LIA
104     STA(ATtmp2)
105     LDB(ATp_array_begin)
106     LDA(ATtmp1)
107     SIA
108     LDB(ATp_left)
109     LDA(ATtmp2)
110     SIA
111
112     /* recur */
113     {
114         qsort5(
115             (word_t *) (*V(ATp_array_begin)),
116             (word_t *) (*V(ATp_left)));
117         /* pivot is already sorted */
118         qsort5(
119             (word_t *) ((*V(ATp_left))
120                 + sizeof (word_t)),
121             (word_t *) (*V(ATp_array_end)));
122     }
123


124     qsort5_return:
125         return;
126 }
```

Quelltext „kompilationsschritte/qsort6.c“

```
083         LDA(ATp_left)
084         INC(sizeof (word_t))
085         STA(ATp_left)
086
087         /* swap */
088         LDB(ATp_left)
089         LIA
090         STA(ATtmp1)
091         LDB(ATp)
092         LIA
093         STA(ATtmp2)
094         LDB(ATp)
095         LDA(ATtmp1)
096         SIA
097         LDB(ATp_left)
098         LDA(ATtmp2)
099         SIA
100
101     goto qsort6_loop;
102     qsort6_pool:
103
104     /* swap */
105     LDB(ATp_left)
106     LIA
107     STA(ATtmp1)
108     LDB(ATp_array_begin)
109     LIA
110     STA(ATtmp2)
111     LDB(ATp_array_begin)
112     LDA(ATtmp1)
113     SIA
114     LDB(ATp_left)
115     LDA(ATtmp2)
116     SIA
117
118     /* recur */
119     {
120         qsort6(
121             (word_t *) (*V(ATp_array_begin)),
122             (word_t *) (*V(ATp_left)));
123         /* now the pseudo-local global variables
124            are cluttered by the call */
125         /* pivot is already sorted */
126         qsort6(
127             (word_t *) ((*V(ATp_left))
128                 + sizeof (word_t)),
129             (word_t *) (*V(ATp_array_end)));
130     }
131
132     qsort6_return:
133         return;
134 }
```

6.7 Joy-Assembler-Umschrift

Als letzter Schritt wird das C-Programm in Joy Assembler umgeschrieben, wobei der Stapelrahmen explizit wiederaufgenommen wird und die statische Speicherauslegung an das Ende der Routine verlegt wird, damit `cal @qsort` direkt in den Routinenrumpf springt.
Der rechts abgedruckte Quelltext unterscheidet sich von 5.1 nur in Kommentaren und Leerzeilen.

Quelltext „kompilationsschritte/qsort6.c“	Quelltext „kompilationsschritte/qsort.asm“

```
001  typedef signed long long int word_t;          | 001
002                                                |
003  #define SUB a -= b;                           |
004  #define DEC(val) a -= val;                    |
005  #define INC(val) a += val;                    |
006  #define JNP(lbl) if (a <= 0) goto lbl;        |
007  #define JNN(lbl) if (a >= 0) goto lbl;        |
008  #define LIA a = *((word_t *) b);              |
009  #define SIA *((word_t *) b) = a;              |
010  #define LDA(ptr) a = *((word_t *) (ptr));     |
011  #define LDB(ptr) b = *((word_t *) (ptr));     |
012  #define STA(ptr) *((word_t *) (ptr)) = a;     |
013  #define LSA(o) a = stack[(o) / sizeof (word_t)]; |
014                                                |
015  #define V(pseudoPtr) ((word_t *) (pseudoPtr)) |
016                                                |
017  void qsort6(                                  | 002  qsort:
018      word_t *p_array_begin, word_t *p_array_end | 003
019  ) {                                           |
020      word_t a, b;                              |
021                                                |
022      word_t *RAM = (word_t[7]) { 0, };         |
023      word_t ATp_array_begin = (word_t) RAM++,  |
024             ATp_array_end   = (word_t) RAM++,  |
025             ATpivot         = (word_t) RAM++,  |
026             ATp_left        = (word_t) RAM++,  |
027             ATp             = (word_t) RAM++,  |
028             ATtmp1          = (word_t) RAM++,  |
029             ATtmp2          = (word_t) RAM++;  |
030                                                |
031      word_t *stack = 4 + (word_t[4]) {         |
032          (word_t) 0x00add100,                  |
033          (word_t) p_array_begin,               |
034          (word_t) p_array_end,                 |
035          (word_t) 0x00000000 };                |
036                                                |
037      /* load local values from stack */        | 004       ; load local values from stack
038      LSA(-3 * sizeof (word_t))                 | 005       lsa -12
039      STA(ATp_array_begin)                      | 006       sta @qsort$p-array-begin
040      LSA(-2 * sizeof (word_t))                 | 007       lsa -8
041      STA(ATp_array_end)                        | 008       sta @qsort$p-array-end
042                                                | 009
043      /* base case */                           | 010       ; base case
044      LDA(ATp_array_end)                        | 011       lda @qsort$p-array-end
045      LDB(ATp_array_begin)                      | 012       ldb @qsort$p-array-begin
046      SUB                                       | 013       sub
047      DEC(sizeof (word_t))                      | 014       dec 4
048      JNP(qsort6_return)                        | 015       jnp @qsort$return
049                                                | 016
050      /* initialize */                          | 017       ; initialize
```

Quelltext „kompilationsschritte/qsort6.c“

```
051     LDB(ATp_array_begin)
052     LIA
053     STA(ATpivot)
054     LDA(ATp_array_begin)
055     STA(ATp_left)
056
057     /* loop */
058     LDA(ATp_array_begin)
059     STA(ATp);
060     qsort6_loop:
061         /* loop through array; the first element
062            is chosen as a pivot and thus
063            immediately skipped on the first
064            iteration */
065         LDA(ATp)
066         INC(sizeof (word_t));
067         STA(ATp)
068
069         /* array end */
070         LDA(ATp_array_end)
071         LDB(ATp)
072         SUB
073         JNP(qsort6_pool)
074
075         /* should the element be swapped? */
076         LDB(ATp)
077         LIA
078         LDB(ATpivot)
079         SUB
080         JNN(qsort6_loop)
081
082         /* advance left pointer */
083         LDA(ATp_left)
084         INC(sizeof (word_t))
085         STA(ATp_left)
086
087         /* swap */
088         LDB(ATp_left)
089         LIA
090         STA(ATtmp1)
091         LDB(ATp)
092         LIA
093         STA(ATtmp2)
094         LDB(ATp)
095         LDA(ATtmp1)
096         SIA
097         LDB(ATp_left)
098         LDA(ATtmp2)
099         SIA
100
101     goto qsort6_loop;
102     qsort6_pool:
103
104     /* swap */
105     LDB(ATp_left)
106     LIA
107     STA(ATtmp1)
108     LDB(ATp_array_begin)
109     LIA
110     STA(ATtmp2)
111     LDB(ATp_array_begin)
```

Quelltext „kompilationsschritte/qsort.asm“

```
018     ldb @qsort$p-array-begin
019     lia
020     sta @qsort$pivot
021     lda @qsort$p-array-begin
022     sta @qsort$p-left
023
024     ; loop
025     lda @qsort$p-array-begin
026     sta @qsort$p
027     qsort$loop:
028         ; loop through array; the first element
029         ; is chosen as a pivot and thus
030         ; immediately skipped on the first iteration
031         lda @qsort$p
032         inc 4
033         sta @qsort$p
034
035         ; array end
036         lda @qsort$p-array-end
037         ldb @qsort$p
038         sub
039         jnp @qsort$pool
040
041         ; should the element be swapped?
042         ldb @qsort$p
043         lia
044         ldb @qsort$pivot
045         sub
046         jnn @qsort$loop
047
048         ; advance left pointer
049         lda @qsort$p-left
050         inc 4
051         sta @qsort$p-left
052

053         ; swap
054         ldb @qsort$p-left
055         lia
056         sta @qsort$tmp1
057         ldb @qsort$p
058         lia
059         sta @qsort$tmp2
060         ldb @qsort$p
061         lda @qsort$tmp1
062         sia
063         ldb @qsort$p-left
064         lda @qsort$tmp2
065         sia
066
067     jmp @qsort$loop
068     qsort$pool:
069
070     ; swap
071     ldb @qsort$p-left
072     lia
073     sta @qsort$tmp1
074     ldb @qsort$p-array-begin
075     lia
076     sta @qsort$tmp2
077     ldb @qsort$p-array-begin
```

Quelltext „kompilationsschritte/qsort6.c“	Quelltext „kompilationsschritte/qsort.asm“

```c
112     LDA(ATtmp1)
113     SIA
114     LDB(ATp_left)
115     LDA(ATtmp2)
116     SIA
117
118     /* recur */
119     {
120         qsort6(
121             (word_t *) (*V(ATp_array_begin)),
122             (word_t *) (*V(ATp_left)));
123         /* now the pseudo-local global variables
124            are cluttered by the call */
125         /* pivot is already sorted */
126         qsort6(
127             (word_t *) ((*V(ATp_left))
128                 + sizeof (word_t)),
129             (word_t *) (*V(ATp_array_end)));
130     }
131

132     qsort6_return:
133         return;
134 }
```

```
078     lda @qsort$tmp1
079     sia
080     ldb @qsort$p-left
081     lda @qsort$tmp2
082     sia
083
084     ; recur
085         lda @qsort$p-left
086         inc 4
087         psh
088         lda @qsort$p-array-end
089         psh
090
091         lda @qsort$p-array-begin
092         psh
093         lda @qsort$p-left
094         psh
095
096         cal @qsort
097         pop
098         pop
099         ; now the pseudo-local global
100         ; variables are cluttered by the call
101
102         ; pivot is already sorted
103
104         cal @qsort
105         pop
106         pop
107
108     qsort$return:
109         ret
110
111     qsort$p-array-begin:
112         data [1]
113     qsort$p-array-end:
114         data [1]
115     qsort$pivot:
116         data [1]
117     qsort$p-left:
118         data [1]
119     qsort$p:
120         data [1]
121     qsort$tmp1:
122         data [1]
123     qsort$tmp2:
124         data [1]
```

Inhaltsverzeichnis

1 Architektur **1**

2 Simulation **1**

3 Sortieralgorithmen **2**
3.1 Quicksort 2
3.2 Shellsort 3
3.3 Vergleich beider Sortierverfahren und ihrer Varianten 3
3.4 Abbildungen 5
3.4.1 Kleines Regime ($0 \leq n \leq 16$) 5
3.4.2 Mittleres Regime ($0 \leq n \leq 32$) 6
3.4.3 Großes Regime ($0 \leq n \leq 64$) 7
3.4.4 Riesiges Regime ($0 \leq n \leq 1024$) 8
3.4.5 Riesiges Regime ohne Insertionsort ($0 \leq n \leq 1024$) 9

4 Mikroinstruktionen **10**
4.1 Mikroinstruktionen (Abschnitt *nop*) 10
4.2 Mikroinstruktionen (Abschnitt *memory*) 10
4.3 Mikroinstruktionen (Abschnitt *jumps*) 11
4.4 Mikroinstruktionen (Abschnitt *stack*) 12
4.5 Mikroinstruktionen (Abschnitt *reg. A*) 14
4.6 Mikroinstruktionen (Abschnitt *reg. A, B*) 15
4.7 Mikroinstruktionen (Abschnitt *i/o*) 15
4.8 Mikroinstruktionen (Abschnitt *rnd*) 16
4.9 Mikroinstruktionen (Abschnitt *hlt*) 16

5 Quelltext **17**
5.1 Quicksort 17
5.2 Shellsort 19

6 Menschliche Kompilation **22**
6.1 C-Konstrukte auflösen 22
6.2 Typsemantik auflösen 23
6.3 `if`-Ausdrücke auflösen 24
6.4 Ausdrücke linearisieren 25
6.5 Präprozessormakros für Joy-Assembler-Instruktionen 27
6.6 Stapelemulierung 29
6.7 Joy-Assembler-Umschrift 32