

Assembler

#006

```

1 gcd:
2 CMP    R0, R1           ; Führt einen Vergleich zwischen R0 und R1
3                                     ; durch. Entsprechend des Vergleichs werden
4                                     ; die Condition-Code-Flags im CPSR gesetzt.
5 SUBGT   R0, R0, R1       ; Führt die Berechnung R0  $\leftarrow$  R0 - R1
6                                     ; durch, falls das Z-Flag gelöscht ist und N- und V-Flag
7                                     ; gleich sind (entspricht nach Tabelle dem '>'),
8                                     ; also R0 > R1
9 SUBLT   R1, R1, R0       ; Führt die Berechnung R1  $\leftarrow$  R1 - R0
10                                    ; durch, falls N- und V-Flag ungleich sind (entspricht nach
11                                    ; Tabelle dem '<'), also R0 < R1
12 BNE     gcd             ; Führt einen Sprung zu der Marke gcd durch,
13                                     ; wenn das Z-Flag nicht gesetzt ist
14                                     ; (entspricht nach Tabelle dem '!='),
15                                     ; also R0  $\neq$  R1

```

Assembler

#013

```
1 BIC      R0, R0, #0x00018000 ; Bits 16 und 15 werden gelöscht,  
2                      ; alle anderen Bits bleiben unverändert.
```

Assembler

#014

```
1 EOR      R0, R0, #0x01800000 ; Bits 23 und 24 werden geändert, 0 ↔ 1  
2                      ; alle anderen Bits bleiben unverändert.
```

Assembler

#015

```

1 TST    R0, #0x00000100      ; prüfe Bit 8 in R0
2 ADDNE  R0, R0, R1        ; berechne R0 ← R0 + R1, falls Bit 8 gesetzt
3
4 TST    R0, #0x00000180      ; prüfe Bit 7 und 8 R0
5 ADDNE  R0, R0, R1        ; berechne R0 ← R0 + R1, falls Bit 7 oder Bit 8 gesetzt
6
7 TST    R0, #0x00000300      ; prüfe Bit 8 und 9 R0
8 ADDEQ  R0, R0, R1        ; berechne R0 ← R0 + R1, falls Bit 8 und Bit 9 nicht gesetzt

```

Assembler

#016

```

1 AND      R1, R0, #0x00000003 ; Bit 0 und 1 ausblenden
2 CMP       R1, #0x00000002 ; prüfe, ob Bit 0 = 0 und Bit 1 = 1
3 ADDEQ    R5, R6, R7    ; berechne  $R5 \leftarrow R6 + R7$ , falls die Bedingung erfüllt ist

```

Befehl	Bedeutung	Operation
UMULL	Multiplikation (vorzeichenlos)	$(RdHi, RdLo) \leftarrow Rm \cdot Rs$
SMULL	Multiplikation (mit Vorzeichen)	$(RdHi, RdLo) \leftarrow Rm \cdot Rs$
UMLAL	Multiplizieren und akkumulieren (vorzeichenlos)	$(RdHi, RdLo) \leftarrow Rm \cdot Rs + (RdHi, RdLo)$
SMLAL	Multiplizieren und akkumulieren (mit Vorzeichen)	$(RdHi, RdLo) \leftarrow Rm \cdot Rs + (RdHi, RdLo)$

Tab. 9: Mnemonik der langen Multiplikationsbefehle der MAC-Unit und die zugehörigen Operationen

Assembler

#021

```
1 MVN      R3, #0x2          ; R3 ← -3  
2 MOV      R3, #0x8000       ; R4 ← 32768  
3 UMULL   R1, R2, R3, R4
```

Das Ergebnis in **R1** (low) bzw. **R2** (high) ist gleich `0xFFFFE8000` bzw. `0x00007FFF`, was wiederum nicht korrekt ist.

Assembler

#024

```

1 MOV    R0, #0          ; S ← 0
2
3 next:
4 ADD    R0, R0, R1      ; S ← S + N
5 SUBS   R1, R1, #1      ; N ← N - 1
6 BNE    next            ; wenn N ≠ 0, dann nochmal
7
8 done:                  ; fertig

```

Tab. 10 zeigt die möglichen Typangaben jeweils mit einem Beispiel. Bei diesen Beispielen wird je-

Datum	Typ	Beispiel	Auffüllen
Wort	-	LDR R1, [R2]	-
Halbwort (unsigned)	H	LDRH R1, [R2]	von links mit 0
Halbwort (signed)	SH	LDRSH R1, [R2]	von links mit dem Vorzeichen
Byte (unsigned)	B	LDRB R1, [R2]	von links mit 0
Byte (signed)	SB	LDRSB R1, [R2]	von links mit dem Vorzeichen

Tab. 10: Mögliche Typenangaben beim Ladebefehl **LDR** für unterschiedliche Datengröße mit und ohne Vorzeichen

Datum	Typ	Beispiel
Wort	-	STR R1, [R2]
Halbwort	H	STRH R1, [R2]
Byte	B	STRB R1, [R2]

Tab. 12: Mögliche Typenangaben beim Speicherbefehl **STR** für unterschiedliche Datengröße



Assembler	#027
1 LDR R1, [R2, #-4] 2 LDRB R3, [R2, #0x200]	

liest für **R2** gleich $0x2000$ in **R1** die 4 Bytes ab Adresse $0x1FFC$ und in **R3** ein Byte von Adresse $0x2200$.

Bei der indizierten Adressierung kann man den Index als Registerwert angeben. Der Index **Rm** darf zusätzlich eine Anweisung an den Barrel-Shifter enthalten. Die Anzahl der zu verschiebenden Bits erfolgt unmittelbar. Die Basisadresse bleibt unverändert.

Nachfolgend ein Beispiel zur indizierten Adressierung. Der Befehl

Assembler	#028
1 LDR R1, [R2, -R3, LSR #4]	

liest für **R2** gleich $0x2000$ und für **R3** gleich $0x0403$ in **R1** ein Byte von Adresse $0x1FC0$. Der Index ist hier gleich $-(R3 \div 16)$.

Neben den im Abschnitt 4.5.1 aufgezeigten (allgemeinen) Adressierungsarten verfügt der ARM7 zwei weitere besondere Formen, der indizierten Adressierung, nämlich

- die Adressierungsart **pre-indexed** und
- die Adressierungsart **post-indexed**.

Die Adressierungsart **pre-indexed** ist im Kern mit der indizierten Adressierung identisch, speichert jedoch zusätzlich die **EA** in das Basisregister **Rn** zurück. Beim Laden eines Datum gilt

$$\begin{aligned} EA &:= Rn + Index \\ Rd &\leftarrow (EA) \\ Rn &\leftarrow EA \end{aligned}$$

bzw.

$$\begin{aligned} EA &:= Rn + Index \\ (EA) &\leftarrow Rd \\ Rn &\leftarrow EA \end{aligned}$$

beim Schreiben eines Datums. Es erfolgt die Indizierung (Basisadresse + **Index**) vor dem Speicherzugriff. Die eckigen Klammern symbolisieren den Speicherzugriff.

Die Syntax der Adressierungsart **pre-indexed** lautet wie folgt.

<i>Befehl {Bedingung} {Typ}</i>	<i>Rd, [Rn, index] !</i>	⟨S ₁₃ ⟩
---------------------------------	--------------------------	--------------------

Man beachte im Vergleich zu ⟨S₁₂⟩ das Ausrufezeichen **!** am Ende der Anweisung. Allein dieses Symbol entscheidet darüber, ob es sich um die Adressierungsart **pre-indexed** handelt.

Nachfolgend zwei Beispiele für **R2** jeweils gleich $0x2000$ und **R3** jeweils gleich $0x0403$.

Assembler	#029
1 LDRSB R1, [R2, R3]! ; EA := R2 + R3, R1 ← [EA], R2 ← EA 2 LDR R1, [R2, -R3, LSR #4]! ; EA := R2 - R3 ÷ 16, R1 ← [EA], R2 ← EA	

Die erste Anweisung überträgt genau ein Byte vorzeichenrichtig von Adresse $0x2403$ nach **R1** und überschreibt **R2** mit $0x2403$. Die zweite Anweisung überträgt 4 Bytes ab Adresse $0x1FC0$ nach **R1** und überschreibt **R2** mit $0x1FC0$.

Die Adressierungsart **post-indexed** überschreibt ebenfalls das Basisregister **Rn**, doch ist die Berechnung unterschiedlich. Beim Laden eines Datum gilt

$EA := Rn$
 $Rd \leftarrow (EA)$
 $Rn \leftarrow EA + Index$

bzw.

$EA := Rn$
 $(EA) \leftarrow Rd$
 $Rn \leftarrow EA + Index$

beim Schreiben eines Datums. Es erfolgt die Indizierung (Basisadresse + *Index*) nach dem Speicherzugriff. Die eckigen Klammern symbolisieren erneut den Speicherzugriff.

Die Syntax der Adressierungsart *post-indexed* lautet wie folgt.

Befehl{Bedingung}{Typ} *Rd, [Rn], Index* $\langle S_{14} \rangle$

Beim Arbeiten mit den einzelnen Adressierungsarten ist folgende Daumenregel hilfreich. *Alles was zwischen eckigen Klammern steht symbolisiert die EA, d. h. die Adresse, auf die der Speicherzugriff stattfindet.*

Nachfolgend zwei Beispiele für **R2** jeweils gleich $0x2000$ und **R3** jeweils gleich $0x0403$.

Assembler		#030
1 LDRSB R1, [R2], R3 ; EA := R2, R1 $\leftarrow [EA]$, R2 $\leftarrow EA + R3$	2 LDR R1, [R2], -R3, LSR #4 ; EA := R2, R1 $\leftarrow [EA]$, R2 $\leftarrow EA - R3 \div 16$	

Die erste Anweisung überträgt genau ein Byte vorzeichenrichtig von Adresse $0x2000$ nach **R1** und überschreibt **R2** mit $0x2403$. Die zweite Anweisung überträgt 4 Bytes ab Adresse $0x2000$ nach **R1** und überschreibt **R2** mit $0x1FC0$.

Tab. 14 bietet einige Übungsbeispiele zu den unterschiedlichen Adressierungsarten.

		+0	+1	+2	+3	+4	+5	+6	+7
Speicherinhalt		0x05	0x06	0x07	0xF0	0x08	0xF1	0x09	0x0A
Register	Wert								
R2	0x20000	Auch in R2 Zurück schreiben							
R3	0x20001								
R4	0x20002								
R5	0x20003								
		Befehl							
		LDR R1, [R2], #-4							
		LDRB R1, [R2, R3, LSR #16]!							
		LDRSH R1, [R3, #1] !							vgl. obsidian
		LDR R1, [R4, #-2] !							write back in R4 !!!
		LDR R1, [R4]							
		LDRH R1, [R2], R3							Beim zwiten mal kein Zugriff
		LDR R1, [R2], -R3, LSR #4							

Tab. 14: Übungsbeispiele für unterschiedliche Adressierungsarten. Welchen Wert haben **R1** bis **R4** nach der Ausführung des Befehls?

4.5.5 Direkte Addressierung

Alle bislang vorgestellten Adressierungstechniken ermitteln die Adresse für den Speicherzugriff über Register. Gibt es eine Möglichkeit, die Adresse direkt mit dem Befehl anzugeben, z. B. durch

Um den Einsatz der Lade- und Speicherbefehle zu vertiefen, folgt nun ein weiteres Beispielprogramm. Dieses Programm soll aus einem Feld von 9 vorzeichenbehafteten 8-Bit Zahlen die kleinste Zahl finden. Erneut wird von einer Lösung in der Programmiersprache C ausgegangen

Anschließend wird eine Registerbelegung gewählt, nämlich

- das Register **R0** für die Laufvariable *n* in `index`,
- das Register **R1** für das (bisherige) Ergebnis *R* in `result`,
- das Register **R2** für die Startadresse des Feldes `array` und
- das Register **R3** als temporäre Variable *t* in `temp`.

Damit ergibt sich folgendes (optimierte) Assembler Programm.

Assembler		#036
1	MOV R0, #9	; R0 : $n \leftarrow$ Anzahl der Elemente (hier 9)
2	MOV R1, #0x7F	; R1 : $R \leftarrow$ max. vorz. 8-Bit Zahl
3	LDR R2, =myArray	; R2 : Adresse des Feldes (zeigt auf a_0)
4	next:	
5	LDRSB R3, [R2], #1	; $t \leftarrow a_i, i \leftarrow i + 1$
6	CMP R3, R1	; Vergleiche R mit t
7	MOVLT R1, R3	; Wenn kleiner, dann übernehmen
8	SUBS R0, R0, #1	; $n \leftarrow n - 1$
9	BNE next	; wenn $n > 0$, dann nochmal
10		
11	MOV R0, R1	; Rückgabe von R immer in R0
12	MOV PC, LR	

Befehl{Bedingung}{Modus} Rn{!}, Liste{^}

⟨S₁₇⟩

Hierbei stehen

- **Befehl** für die mnemonische Kennzeichnung des Befehls (**LDM** (*load multiple registers*) bzw. **STM** (*store multiple registers*)),
- **Bedingung** die mnemonische Kennzeichnung der Ausführungsbedingung (optional, s. Tab. 2 auf Seite 28),
- **Modus** (optional) für die Art, wie die Basisadresse verwendet wird (**IA**, **IB**, **DA** oder **DB**),
- **Rn** für die Basisadresse (darf nicht der Programmzähler **PC** sein) und
- **Liste** für die Angabe der zu speichernden/ladenden Register.

In diesen Anweisungen kann **Liste** die Register **R0** bis **R12**, **SP**, **LR** und **PC** umfassen, wobei die Register durch Komma getrennt sind, z.B. **R4**, **LR**, oder durch Angaben eines Bereiches, z.B. **R0-R4**, angegeben werden können. Die Angabe der Register wird durch geschweifte Klammern eingeschlossen, z.B. **{R0-R4, PC}**.

Die Adresse kann

- nach jedem Transfer inkrementiert werden (**Mode** ist gleich **IA**, *increment after*),
- vor jedem Transfer inkrementiert werden (**Mode** ist gleich **IB**, *increment before*),
- nach jedem Transfer dekrementiert werden (**Mode** ist gleich **DA**, *decrement after*) oder
- vor jedem Transfer dekrementiert werden (**Mode** ist gleich **DB**, *decrement before*).

Das Basisregister kann entweder unverändert belassen werden oder es wird aktualisiert, um auf die nächste Adresse im Speicherblock zu verweisen. Letzteres wird als *write back* bezeichnet und wird in der Syntax durch das Ausrufezeichen ! nach **Rn** dargestellt.

Assembler

#037

```
1 LDM      R8, {R0, R2, R9}      ; LDM is a synonym for LDMIA
2 STMDB    R1!, {R3-R6, R11, R12}
```

Falls sich der Prozessor in einem (privilegierten) Unterbrechungsbetriebsart befindet, kann man statt der Register der entsprechenden Betriebsart auch die Register der User-Modus-Betriebsart lesen oder schreiben. In der Syntax wird dies durch ^ nach **Liste** angezeigt. Falls dann der Befehl **LDM** ausgeführt wird und der Befehlszähler **PC** in **Liste** enthalten ist, wird ein zusätzlicher Transfer vom **SPSR** in das **CPSR** durchgeführt. Dies ist eine nützliche Ergänzung beim Rücksprung aus einer **ISR** oder zur Realisierung eines Kontextwechsels (*context switch*).

Die Reihenfolge der Lade- und Schreibeoperationen sind hierbei fest vorgegeben. Es werden immer die Registerinhalte mit **R0** beginnend gelesen.

Lade- und Speicherbefehle für mehrere Register können ganze Speicherblöcke kopieren. Dies zeigt das nachfolgende Beispiel.

Assembler

#038

```
1 LDR      R0, =src          ; R0 ← pointer to source block
2 LDR      R1, =dst          ; R1 ← pointer to destination block
3 copy_block:
4 LDM      R0, {R4-R11}      ; Load 8 words from the source
5 STM      R1, {R4-R11}      ; and put them at the destination
```

Hier hat man mit zwei Instruktionen 32 Bytes kopiert.

Befehl	vgl.	cond	Kodierung
ADD R0, R1, R2	Abb. 13	always	0xE0810002
ADDS R0, R1, R2	Abb. 13	always	0xE0910002
ADDEQ R3, R4, R5	Abb. 13	EQ	0x00843005
ADDNE R6, R7, R8, LSL #1	Abb. 13	NE	0x10876088
SUBGT R6, R7, R8, LSR #2	Abb. 13	GT	0xC0476128
SUBGT R6, R7, R8, LSR R9	Abb. 13	GT	0xC0476918
SUBGE R6, R7, R8, LSR #4	Abb. 13	GE	0xA0476228
MOV R6, R8, ROR R9	Abb. 13	always	0xE1A06978
TST R7, R8, ASR R9	Abb. 13	always	0xE1170958
MUL R1, R2, R3	Abb. 19	always	0xE0010392
MLA R1, R2, R3, R4	Abb. 19	always	0xE0214392
LDR R1, [R2]	Abb. 21	always	0xE5921000
LDR R3, [R4, #4]	Abb. 21	always	0xE5921004
LDRB R1, [R2, #1]	Abb. 21	always	0xE5D21001
LDRB R1, [R2, #1]!	Abb. 21	always	0xE5F21001
LDRB R1, [R2, #1]	Abb. 21	always	0xE4521001
LDRSB R1, [R2, #1]	Abb. 21	always	0xE1D210D1
LDRH R1, [R2, #2]	Abb. 21	always	0xE1D210B2
LDRSH R1, [R2, R3]	Abb. 21	always	0xE19210F3
LDR R1, [R2, R3, LSL #2]	Abb. 21	always	0xE7921103
LDRB R1, [R2, R3, LSL #2]	Abb. 21	always	0xE7D21103
LDRH R1, [R2, R3, LSL #2]	Abb. 21	Achtung, böse Falle!	

Tab. 17: Beispiele zur Kodierung bzw. Dekodierung von ARMv6-Instruktionen

Assembler	#055
1	
2 EOR R0, R0, R1	; R0: U, R1: V ; R0 \leftarrow U XOR V
3 EOR R1, R0, R1	; R1 \leftarrow R0 XOR V = $(U \text{ XOR } V) \text{ XOR } V = U$
4 EOR R0, R0, R1	; R0 \leftarrow R0 XOR R1 = $(U \text{ XOR } V) \text{ XOR } U = V$
5	; R0 = V, R1 = U

4.10.1.2 Bytereihenfolge vertauschen

In Abschnitt 4.8.6 wurde bereits ein Befehl vorgestellt, mit dem man die Bytereihenfolge in einem Wort vertauschen kann. Der dazu benötigte Befehl **REV** ist jedoch erst ab der Architektur ARMv6 vorhanden. In den älteren Architekturen musste man diese Funktion nachbilden.

Assembler	#056
1 emulate_rev:	
2 EOR R3, R1, R1, ROR #16	
3 BIC R3, R3, #0x00FF0000	
4 MOV R0, R1, ROR #8	
5 EOR R0, R0, R3, LSR #8	

4.10.1.3 Vorzeichengleichheit prüfen

```

1 ; R0 : U, R1: V
2 MOV      R2, R0      ; R2 ← U
3 MOV      R0, R1      ; R0 ← V
4 MOV      R1, R2      ; R1 ← R2 = U
5          ; R0 = V, R1 = U

```

Wenn jedoch kein freies Register zur Verfügung steht, ist dies nicht mehr so einfach, doch es ist möglich.

Seien zwei Zahlen in 2-er-Komplementdarstellung gegeben. Wie kann man möglichst einfach prüfen, ob beide Zahlen gleiche oder ungleiche Vorzeichen haben? Da das höchstwertigste Bit das Vorzeichen darstellt, kann man dies durch eine XOR-Verkündigung realisieren.

```

1 EORS    R2, R0, R1      ; R0: a, R1: b
2 BPL     equal_sign

```

4.10.1.4 Prüfen ob Zweierpotenz

Manchmal muss man feststellen, ob ein Zahlenwert (größer 0) eine Zweierpotenz darstellt. Wenn ein Wert eines Registers gleich einer Zweierpotenz 2^n ist, so ist ausschließlich das n -te Bit gesetzt (0-relativ gezählt). In der binären Darstellung von $2^n - 1$ sind nur die niederwertigsten $n - 1$ Bits gleich 1. Diese Eigenschaften kann man ausnutzen.

```

1 ADD    R1, R0, #1        ; check R0 if power of two, R0>0
2 TST    R0, R1
3 BEQ    is_power_of_two

```

Die Anweisung **TST** führt eine bit-weise Und-Verkündigung der beiden Operanden aus (wie **ANDS**, jedoch wird das Ergebnis verworfen).

4.10.1.5 Parität berechnen

Die Parität eines 32-Bit-Wortes ist die XOR-Verknüpfung aller 32 Bits. Sei $R = \sum_{i=0}^{31} r_i 2^i$ der Wert eines Registers, so ist

$$P(R) := r_0 \oplus r_1 \oplus \dots \oplus r_{31}$$

die Parität dieses Wertes (\oplus stellt die logische XOR-Operation dar). Die Parität lässt sich mit nur 5 Befehlen berechnen.

```

1 ; Berechne die Parität von R0
2 EOR    R1, R0, R0 LSR #1
3 EOR    R1, R1, R1 LSR #2
4 EOR    R1, R1, R1 LSR #4
5 EOR    R1, R1, R1 LSR #8
6 EOR    R1, R1, R1 LSR #16

```

Das Paritätsbit von **R0** befindet sich anschließend im niederwertigsten Bit von **R1** (wieso?).

Aber auch die Multiplikation mit $2^n + 1$ kann effizient implementiert werden. Es gilt nämlich $x \cdot (2^n + 1) = x \cdot 2^n + x$, und dies kann durch eine Addition realisiert werden. Hierbei ist x der erste Operand und $x \cdot 2^n$ bildet den zweiten Operanden (*shifter operand*).

Assembler

#061

```
1 ADD R1, R0, R0, LSL #4 ; R1 ← R0 + R0 · 24 = R0 · 17
```

Jetzt ist es nicht schwer, die Multiplikation mit $2^n - 1$ zu implementieren. Es gilt $x \cdot (2^n - 1) = x \cdot 2^n - x$, und dies kann durch eine Subtraktion realisiert werden. Hierbei ist x wiederum der erste Operand und $x \cdot 2^n$ bildet den zweiten Operanden (*shifter operand*), d. h. es muss der Befehle **RSB** (umgekehrte Subtraktion) verwendet werden.

Assembler

#062

```
1 RSB R1, R0, R0, LSL #4 ; R1 ← R0 · 24 - R0 = R0 · 15
```

Multiplikationen eines 64-Bit-Wertes x mit einer Zweierpotenz 2^n (Ergebnis ebenfalls 64 Bit) kann man durch mehrere Schiebeoperationen zusammenfügen. Seien L (*low*) und H (*high*) die unteren bzw. oberen 32 Bits des 64-Bit-Wertes. Es gelten dann

4.10.1.7 Vorzeichenrichtige Erweiterung

Beim Lesen der Register von Peripherieeinheiten via Memory-mapped-I/O kommt es vor, dass man Daten liest, die weder 8, noch 16 noch 32 Bits breit sind. Sind die Werte in der 2-er-Komplement-Darstellung gegeben, so müssen sie von $n \neq 32$ auf 32 Bits vorzeichenrichtig erweitert werden, die sog. *sign(ed) extension*. Es folgt ein Beispiel für $n = 12$.

Assembler

#063

```
1 MOV R1, R1, LSL #20 ; 12 Bits linksbündig anordnen
2 MOV R1, R2, ASR #20 ; 12 Bits wieder rechtsbündig anordnen,
3 ; aber mit Vorzeichen auffüllen
```

Der Compiler verwendet diesen Mechanismus, um z.B. nachfolgenden C-Source-Code in Assembler-Code zu übersetzen.

4.10.1.8 Mittelwert berechnen

Die Berechnung des Mittelwertes zweier ganzen Zahlen kann schon mal häufiger in einem Programm vorkommen. Die erste Implementierung dieser Aufgabe könnte wie folgt aussehen.

Assembler

#064

```
1 ; R0 : a, R1 : b, Ergebnis nach R2
2 ADD R2, R0, R1 ; R2 ← (a + b) mod 232
3 MOV R2, LSR #1 ; R2 ← R2 ÷ 2 (unsigned)
```

So würde auch ein C-Compiler eine Anweisung der Form $y = (a + b) / 2^0$ übersetzen! Dieser Algorithmus würde jedoch nicht immer das korrekte Ergebnis berechnen. Wenn die Summe der Werte die Obergrenze $2^{32} - 1$ (bei `unsigned`, 32 Bits) überschreitet ist das Ergebnis falsch. Das Verhalten eines C-Programms bei solchen Bereichsüberschreitungen (`signed` wie auch `unsigned`) ist undefiniert. Die meisten Compiler für C/C++ erzeugen Code, der Bereichsüberschreitungen schlichtweg ignoriert.

Was Bereichsüberschreitungen passieren kann, zeigt eindrucksvoll die Geschichte des Erstflugs der europäischen Trägerrakete Ariane 5, am 4. Juni 1996, obwohl die dort verwendete Software in Ada⁴⁶ programmiert wurde. Gleichwohl muss hierzu bemerkt werden, dass der Fehler letztendlich durch ein mangelhaftes System-Design und vor allem ein falsches Projekt-Management (Verzicht auf den Systemtest der betroffenen Komponenten aus Kostengründen) geradezu provoziert wurde (s. [Flight 501 Failure](#), Bericht der Untersuchungskommission).

Zur korrekten Berechnung des Mittelwertes (`unsigned`) würde man die Addition anpassen.

Assembler

#065

```
1 ; R0 : a, R1 : b, Ergebnis nach R2
2 ADDS R2, R0, R1 ; R2 ← (a + b) mod 232, C ← (a + b) ÷ 232
3 MOV R2, LSR #1 ; R2 ← R2 ÷ 2, Bit 31 ist (noch) 0
4 ORRCS R2, #0x80000000 ; Bit 31 kopieren
```

Durch Berechnung des 33-Bit-Ergebnisses (inkl. C-Flag) kann man das Ergebnis geeignet anpassen. Noch effektiver ist jedoch folgende Implementierung.

Assembler

#066

```
1 ; R0 : a, R1 : b, Ergebnis nach R2
2 ADDS R2, R0, R1 ; R2 ← (a + b) mod 232, C ← (a + b) ÷ 232
3 MOV R2, R2, RRX ; R2 ← (C, R2) ÷ 2
```

Jedoch kann z. B. die Zahl -1 ($0xFFFFFFFF$) nicht mit dem Befehl **MOV** geladen werden. Hierzu kann man jedoch den Befehl **MVN** verwenden.

Assembler	#068
1 MVN R0, #0 ; $R0 \leftarrow -1$	

Der (eigentlich ungültige) Befehl

4.10.1.10 Minimum, Maximum oder Betrag berechnen

Anweisungen der Form

dienen der Berechnung des Maximums oder Minimums zweier Variablen. Anweisung dieser Art sind häufig in Programmen zu finden. Sie lassen sich effizient durch ARM-Befehle realisieren. Die Berechnung des Maximums erfolgt durch folgende Anweisungen.

Assembler	#071
1 CMP R0, R1 ; berechne $R0 - R1$	
2 MOVG R2, R0 ; $R2 \leftarrow R0$, wenn $R0 - R1 > 0$, d. h. wenn $R0 > R1$	
3 MOVLE R2, R1 ; $R2 \leftarrow R1$, wenn $R1 \geq R0$	

Die Berechnung des Minimums erfolgt durch folgende Anweisungen.

Assembler	#072
1 CMP R0, R1 ; berechne $R0 - R1$	
2 MOVLT R2, R0 ; $R2 \leftarrow R0$, wenn $R0 - R1 < 0$, d. h. wenn $R0 < R1$	
3 MOVGE R2, R1 ; $R2 \leftarrow R1$, wenn $R1 \leq R0$	

Assembler	#073
1	;
2 CMP R0, #1 ; $R0: x$, $R1: y$, $R2: z$;
3 CMPNE R1, #3 ; $x = 1$;
4 CMPNE R2, #5 ; $y = 3$ prüfen, wenn $x \neq 1$;
5 BNE done ; $z = 5$ prüfen, wenn $x \neq 3$ und $y \neq 3$;
6	;
7 done:	;
	;
	;
	;
	;
	;

Assembler	#074
1	;
2 CMP R0, #2 ; $R0: u$, $R1: v$, $R2: w$;
3 CMPEQ R1, #4 ; $u = 2$;
4 CMPEQ R2, #6 ; $v = 4$ prüfen, wenn $u = 2$;
5 BNE done ; $w = 6$ prüfen, wenn $u = 2$ und $v = 4$;
6	;
7 done:	;
	;
	;
	;
	;

C/C++

#022

```

1 #define TYPE           unsigned
2
3 int search(TYPE const array[], int const n /* > 0 */, TYPE const key)
4 {
5     while ( --n >= 0 )
6     {
7         if ( array[n] == key ) break;
8     }
9
10    return n;
11 }
```

Diese Funktion findet den (letzten) Eintrag vom Wert `key` in dem Feld `array`, indem sie das Feld von hinten nach vorne durchsucht. Sie gibt den Feldindex des gefundenen Eintrags zurück oder `-1`, wenn der Eintrag nicht gefunden wurde. Dadurch, dass der Feldindex auf 0 runterzählt, kann man die Schleife effizient implementieren.

Assembler

#075

```

1 search:                      ; R0: array, R1: n, R2: key, n ≥ 0
2 SUBS   R1, R1, #1            ; n ← n - 1
3 BLT    done                 ; fertig, wenn n < 0
4 LDR    R3, [R0, R1, LSL #2]  ; R3 ← array[n]
5 CMP    R3, R2                ; vergleiche R3 mit key
6 BNE    next                 ; falls nicht gleich, dann wiederholen
7 done:                         ; R0: Rückgabe
8 MOV    R0, R1                ; R0: Rückgabe
9 MOV    PC, LR                ; return i
```

Wenn jedoch der erste Eintrag gefunden werden muss, muss man das Feld von vorne nach hinten durchsuchen.

C/C++

#023

```

1 #define TYPE           unsigned
2
3 int search(TYPE const * array, int const n /* >= 0 */, TYPE const key)
4 {
5     int i = 0;
6
7     while ( i++ < n )
8     {
9         if ( *array++ == key )
10            return i;
11
12        i++;
13    }
14
15    return -1;
16 }
```

Assembler

#076

```

1 search:                      ; R0: array, R1: n, R2: key, n ≥ 0
2 MOV    R4, R0                ; R4 ← array
3 MOV    R0, #0                 ; R0: Laufindex i ← 0
4 next:                         ; R0: Laufindex i
5 CMP    R0, R1                ; vergleiche i mit n
6 ADD    R0, R0, #1            ; i ← i + 1
7 BGE    fail                 ; nichts gefunden, wenn i ≥ n
8 LDR    R3, [R4], #4          ; R3 ← *array++
9 CMP    R3, R2                ; vergleiche R3 mit key
10 BNE   next                 ; falls nicht gleich, dann wiederholen
11 MOV    PC, LR                ; return i
12 fail:                         ; lade -1
13 MVN    R0, #0
14 done:                         ; R0: -1
15 MOV    PC, LR                ; return -1
```

Beim sequentiellen Lesen von Feldeinträgen, die von vorne nach hinten ausgelesen werden, sind folgende Anweisungen sinnvoll.

```

1 LDR      R0, [R1], #4          ; R0 ← *ptr++ für Wörter
2 LDRH    R0, [R1], #2          ; R0 ← *ptr++ für Halbwörter (unsigned)
3 LDRB    R0, [R1], #1          ; R0 ← *ptr++ für Bytes (unsigned)

```

Hierbei zeigt **R1** auf den nächsten Feldeintrag. Die entsprechende Anweisungen wird in einer Schleife durchgeführt.

Beim sequentiellen Lesen von n Feldeinträgen (Feldindex von 0 bis $n-1$), die von hinten nach vorne ausgelesen werden, geht man wie folgt vor. Zunächst positioniert man den Feldzeiger auf das n -te Element (das jedoch nie geladen wird).

```

1 ADD     R0, R0, R2, LSL #2    ; ptr += n für Wörter
2 ADD     R0, R0, R2, LSL #1    ; ptr += n für Halbwörter
3 ADD     R0, R0, R2            ; ptr += n für Bytes

```

Dann führt man in der Schleife n mal einen Ladebefehl aus.

```

1 LDR      R0, [R1, #-4]!       ; R0 ← --ptr für Wörter
2 LDRH    R0, [R1, #-2]!       ; R0 ← --ptr für Halbwörter (unsigned)
3 LDRB    R0, [R1, #-1]!       ; R0 ← --ptr für Bytes (unsigned)

```

Hierbei zeigt **R1** auf den (ggf. bereits gelesenen) Wert hinter dem nächsten Feldeintrag.

Beim sequentiellen Lesen von $n+1$ Feldeinträgen (Feldindex von 0 bis n), die von hinten nach vorne ausgelesen werden, geht man wie folgt vor. Zunächst positioniert man den Feldzeiger wiederum auf das n -te Element. Dann führt man in der Schleife $n+1$ mal einen Ladebefehl aus.

```

1 LDR      R0, [R1], #-4        ; R0 ← *ptr-- für Wörter
2 LDRH    R0, [R1], #-2        ; R0 ← *ptr-- für Halbwörter (unsigned)
3 LDRB    R0, [R1], #-1        ; R0 ← *ptr-- für Bytes (unsigned)

```

Hierbei zeigt **R1** auf den nächsten Feldeintrag.

```

1                                     ; berechne (R0, R1) + (R2, R3)
2                                     ; R0 und R2: niedwertige Wörter (Bits 0...31)
3                                     ; R1 und R3: höherwertige Wörter (Bits 31...63)
4 ADDS    R4, R0, R2              ; addiere die niedwertigen Wörter
5 ADC     R5, R1, R3              ; addiere die höherwertigen Wörter + C-Flag

```

Der Assembler-Code für die 64-Bit-Subtraktion ergibt sich analog.

```

1                                     ; berechne (R0, R1) - (R2, R3)
2                                     ; R0 und R2: niedwertige Wörter (Bits 0...31)
3                                     ; R1 und R3: höherwertige Wörter (Bits 31...63)
4 SUBS    R4, R0, R2              ; subtrahiere die niedwertigen Wörter
5 SBC     R5, R1, R3              ; subtrahiere die höherwertigen Wörter

```

Diese Vorgehensweise lässt sich problemlos auf Operanden mit $n \cdot 32$ Bits erweitern.

Soll ein Wert vom Datentyp `int64_t` und ein Wert vom Datentyp `int32_t` addiert werden, so muss das Vorzeichen korrekt erweitert werden.

```

1                                     ; 64-Bit-Wert x (int64_t) in R0 und R1
2                                     ; 32-Bit-Operand a (int32_t) in R2
3                                     ; x ← x + a
4 ADDS    R0, R0, R2              ; addiere a zum niedwertigen Wort
5 MOV     R2, R2, ASR #31         ; -1 wenn a < 0, sonst 0
6 ADC     R1, R1, R2              ; höherwertiges Wort anpassen

```

Im Gegensatz dazu der Code für einen Operanden vom Typ `uint32_t`.

```

1                                     ; 64-Bit-Wert x (sint64_t) in R0 und R1
2                                     ; 32-Bit-Operand a (uint32_t) in R2
3                                     ; x ← x + a
4 ADDS    R0, R0, R2              ; addiere a zum niedwertigen Wort
5 ADC     R1, R1, #0               ; addiere Übertrag

```

Wenn man **ein Hilfsregister verwendet**, kann man diese Operation jedoch noch weiter optimieren!

Assembler

#086

```

1 ; berechne (R0, R1) LSL 4
2 ; R0: niederwertiges Wort L (Bits 0...31)
3 ; R1: höherwertiges Wort H (Bits 32...63)
4 MOV   R4, R1, LSL #4
5 ORR   R1, R4, R0, LSR #28
6 MOV   R0, R0, LSL #4

```

Diese drei Instruktionen benötigen nur 6 Taktzyklen zur Ausführung.

DIVISION

Der Barrel-Shifter berechnet $\lfloor \frac{\dots}{2^k} \rfloor$ durch die Operation **ASR**. Muss man deshalb bei der Berechnung von $N \div 2^k$ eine Fallunterscheidung durchführen. Dies zeigt nachfolgendes Beispiel an der Division durch 8.

Assembler

#089

```

1 ; berechne R1  $\leftarrow \text{R0} \div 2^3$  (signed)
2 MOVS  R1, R0
3 ADDMI R1, R1, #7
4 MOV   R1, R1, ASR #3
      ; R1  $\leftarrow \text{R0} + 7 = \text{R0} + 7$ , wenn R0 < 0
      ; R1  $\leftarrow \text{R0} \div 8$ 

```

Diese Art der Kodierung kann jedoch Divisionen durch 2^k mit $k > 8$ nicht durchführen, da für $k > 8$ die Zahl $2^k - 1$ nicht als Immediate-Konstante darstellbar ist. Dann ist nachfolgende Implementierung besser.

Assembler

#090

```

1 ; berechne R1  $\leftarrow \text{R0} \div 2^{12}$  (signed)
2 MOVS  R1, R0, ASR #31
3 ADD   R1, R0, R1, LSR #20
4 MOV   R1, R1, ASR #12
      ; R1  $\leftarrow 0$ , wenn R0  $\geq 0$  bzw. R1  $\leftarrow -1 = 0xFFFFFFFF$ , wenn R0 < 0
      ; R1  $\leftarrow \text{R0} + (2^{12} - 1) = \text{R0} + 4095$ , wenn R0 < 0
      ; R1  $\leftarrow \text{R0} \div 4096$ 

```

Zur Implementierung braucht man eine lange Multiplikation mit 64-Bit-Ergebnis.

Assembler

#091

```

1 ; R0 sei durch 5 zu dividieren
2 LDR   R1, =0xCCCCCCCCD
3 UMULL R2, R3, R0, R1
4 MOV   R0, R3, LSR #2
      ; R1  $\leftarrow 0xCCCCCCCCD$ 
      ;  $(\text{R2}, \text{R3}) \leftarrow \text{R0} \cdot \text{R1} \leftarrow \text{R0} \cdot 0xCCCCCCCCD$ 
      ;  $\text{R0} \leftarrow \text{R3} \div 2^2 = (\text{R0} \cdot 0xCCCCCCCCD) \div 2^{34} = \text{R0} \div 5$ 

```

Tab. 18 zeigt die wichtigsten magischen Zahlen zur Division mit Konstanten.

<i>K</i>	<i>Q</i>	<i>n</i>
3	0xAAAAAAAB	1
5	0xCCCCCCCCD	2
9	0x38E38E39	1
10	0xCCCCCCCCD	3
11	0xBA2E8BA3	3
25	0x51EB851F	3
125	0x10624DD3	3

Tab. 18: Magische Zahlen *Q* zur Implementierung der 32-Bit-Division (Datentyp `uint32_t`) mit Konstanten *K* > 0 für den ARM7-Prozessor. Quelle: „*Hacker’s Delight*“ von H. S. Warren, Addison-Wesley (2008)

<i>K</i>	<i>I</i>
3	0xAAAAAAAB
5	0xCCCCCCCD
7	0xB6DB6DB7
9	0x38E38E39
11	0xBA2E8BA3
13	0xC4EC4EC5
15	0xEEEEEEEF
17	0xF0F0F0F1
19	0x286BCA1B
21	0x3CF3CF3D
23	0xE9BD37A7
25	0xC28F5C29
255	0xFFFFEFFF
65 535	0xFFFFFFFF

Tab. 19: Modulare multiplikative Inverse *I* zur 32-Bit-Division mit Konstanten *K* für den ARM7-Prozessor. Quelle: „Hacker’s Delight“ von H. S. Warren, Addison-Wesley (2008)

Somit ist also $0xB6DB6DB7$ das zu 7 inverse Element bei der Multiplikation modulo 2^{32} . Multipliziert man $0xB6DB6DB7$ mit 7 so erhält man $0x50000001$. Modulo 2^{32} ergibt sich der Wert 1.

Zur Implementierung für den ARM7 braucht man nur eine normale Multiplikation mit 32-Bit-Ergebnis.

Assembler	#092
1	<i>; R0</i> (unsigned) sei durch 7 zu dividieren
2 LDR R1, =0xB6DB6DB7	<i>; R1</i> $\leftarrow 0xB6DB6DB7$
3 MUL R3, R0, R1	<i>; R3</i> $\leftarrow R0 \cdot R1 = R0 \cdot 0xB6DB6DB7 = R0 \div 7 \pmod{2^{32}}$

Die Division durch 10 wird durch die Division durch 5 und anschließender Division durch 2 realisiert.

Assembler	#093
1	<i>; R0</i> (unsigned) sei durch 10 zu dividieren
2 LDR R1, =0xCCCCCCCD	<i>; R1</i> $\leftarrow 0xCCCCCCCD$
3 MUL R3, R0, R1	<i>; R3</i> $\leftarrow R0 \cdot R1 = R0 \cdot 0xCCCCCCCD = R0 \div 5 \pmod{2^{32}}$
4 MOV R3, R3, LSR #1	<i>; R3</i> $\leftarrow R3 \div 2 = R0 \div 10$

4.10.7.1 Fakultät

Die Berechnung der Fakultät kann wie folgt durchgeführt werden.

$$n! = \begin{cases} \prod_{i=1}^n i & n > 0 \\ 1 & n = 0 \end{cases}$$

Die Funktion factorial für 32-Bit-Ergebnisse sieht in Assembler wie folgt aus.

Assembler	#094
1 factorial:	
2 MOVS R1, R0	<i>; n</i> ≥ 0
3 MOVEQ R0, #1	<i>; R0</i> $\leftarrow n$ (Schleifenzähler)
4 next:	<i>; wenn n = 0, dann Ergebnis 1</i>
5 SUBNE R1, R1, #1	<i>; wenn n $\neq 0$, dann n $\leftarrow n - 1$</i>
6 MULNE R0, R1, R0	<i>; wenn n $\neq 0$, R0 $\leftarrow R0 \cdot (R0 - 1)$</i>
7 BNE next	<i>; wenn n $\neq 0$, dann nochmal</i>
8 MOV PC, LR	

Wie jedes iterative Verfahren lässt sich auch dieser Algorithmus rekursiv definieren.

$$n! = \begin{cases} n \cdot (n - 1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

Assembler	#095
1 factorial:	<i>; R0</i> $\leftarrow n$, $n \geq 0$
2 STM SP!, {R1, LR}	<i>; R1</i> $\leftarrow n$
3 MOV R1, R0	<i>; R0</i> $\leftarrow n - 1$
4 SUBS R0, R0, #1	<i>; n = 0: R0</i> $\leftarrow 1$
5 MOVLT R0, #1	<i>; n > 0: R0</i> $\leftarrow (n - 1)!$
6 BGE factorial	
7 MULGE R0, R1, R0	<i>; n > 0: R0</i> $\leftarrow R1 \cdot R0 = n \cdot (n - 1)! = n!$
8 LDM SP!, {R1, PC}	

Die diese Routine jedoch die Zwischenergebnisse auf dem Stack zwischenspeichert, ist sie weniger effizient als das iterative Verfahren.

4.10.7.2 Skalarprodukt

Das Skalarprodukt

$$S = \sum_{i=0}^{N-1} a_i b_i$$

spielt bei vielen Anwendungen der digitalen Signalverarbeitung eine wichtige Rolle. Hierbei können die Zahlenwerte a_i und b_i Koeffizienten eines Filters oder Zeitreihen eines Signals darstellen. Beide werden für gewöhnlich in Feldern abgelegt. Angenommen, **R1** steht für die Zahl N ($N > 0$) und die Adressen der Felder von a_i und b_i ($a_i, b_i \in \{-2^{16}; 2^{16} - 1\} \forall i = 0, 1 \dots N - 1$) werden durch **R2** und **R3** angegeben. Wenn (vorab) bekannt ist, dass das Ergebnis als 32-Bit-Wert darstellbar ist, so kann man den Algorithmus wie folgt implementieren.

Assembler

#096

```
1 scalmul:                                ; R0: pointer a to ai
2                                     ; R1: pointer b to bi
3                                     ; R2: number of bytes N number of items
4 MOV    R0, #0                           ; S ← 0
5 next:                                ; R4 ← ai, a++
6 LDRSH R4, [R2], #2                     ; R4 ← ai, a++
```

```
7 LDRSH R5, [R3], #2                  ; R5 ← bi, b++
8 MLA   R0, R4, R5, R0                 ; S ← ai · bi + S
9 SUBS  R1, R1, #1                   ; N ← N - 1 und Flags setzen
10 BNE   next                         ; wiederholen, wenn N ≠ 0
```

Wenn das Ergebnis allgemein als 64-Bit-Wert dargestellt werden soll, muss man die Anweisung **MLA** durch die Anweisung **SMLAL** ersetzen und das Programm entsprechend anpassen.

4.10.7.3 Funktionen mit Fallunterscheidung

Sei die Funktion $f(n)$ wie folgt definiert.

$$f(n) := \begin{cases} \frac{n}{2} & \text{wenn } n \text{ gerade} \\ 3n + 1 & \text{wenn } n \text{ ungerade} \end{cases}$$

Diese Funktion soll implementiert werden, wobei der Übergabeparameter n ($n > 0$) und das Ergebnis $f(n)$ durch **R0** dargestellt werden.

Assembler	#097
<pre> 1 function: 2 TST R0, #1 ; $(n \wedge 1) \equiv (n \bmod 2)$ 3 MOVEQ R0, R0, LSR #1 ; wenn n gerade: R0 $\leftarrow \frac{n}{2}$ 4 ADDNE R0, R0, R0, LSL #1 ; wenn n ungerade: R0 $\leftarrow 2n + n = 3n$ 5 ADDNE R0, R0, #1 ; wenn n ungerade: R0 $\leftarrow \mathbf{R0} + 1 = 3n + 1$ 6 MOV PC, LR </pre>	

Ob n gerade oder ungerade ist, kann man durch Überprüfung des niedrigwertigsten Bits feststellen. Hierzu nutzt man den Befehl **TST**.

4.10.7.4 Polynom auswerten

Zur Berechnung eines Polynoms wird das Horner-Schema verwendet,

$$\begin{aligned} P(x) &= \sum_{i_0}^N p_i x^i \\ &= p_0 + x(p_1 + x(p_2 + \cdots + x(p_{n-1} + p_n x) \cdots)). \end{aligned}$$

Seien die Koeffizienten p_i und x 16-Bit-Zahlen mit Vorzeichen und seien die $N + 1$ Koeffizienten in einem Feld abgelegt. Die Funktion lässt sich, wenn das Ergebnis maximal 32 Bit umfasst, wie folgt implementieren.

Assembler	#098
<pre> 1 horner: 2 MOV R0, R1, LSL #1 ; R0: Zeiger p auf p_i, R1: N, $N > 0$, R2: x 3 LDRSH R3, [R0], #-2 ; R0 $\leftarrow R0 + 2 \cdot R1$, Zeiger auf p_N 4 next: 5 LDRSH R4, [R0], #-2 ; R3: S $\leftarrow p_N$, p dekrementieren 6 MLA R3, R2, R3, R4 ; R4 $\leftarrow p_i$, p dekrementieren 7 SUBS R1, R1, #1 ; S $\leftarrow S \cdot x + p_i$ 8 BGT next ; N $\leftarrow N - 1$ 9 MOV R0, R3 </pre>	

Für 64-Bit-Ergebnisse kann man den Code leicht entsprechend anpassen.

Um den Speicherinhalt eines Datenblockes von N ($N > 0$) Bytes (in **R2**) von der Zieladresse **R0** zu der Bestimmungsadresse **R1** zu kopieren, kann man wie folgt vorgehen.

Assembler

#099

```

1 copy_memory:           ; R0: source pointer s
2                         ; R1: destination pointer d
3                         ; R2: number of bytes N, N > 0
4 next:                 ; R3 ← *s++
5 LDRB      R3, [R0], #1   ; *d++ ← R3
6 STRB      R3, [R1], #1   ; N ← N - 1, set flags
7 SUBS      R2, R2, #1     ; continue, if N ≠ 0
8 BNE       next

```

Um den Speicherinhalt eines Datenblockes von N ($N > 0$) Bytes (in **R2**) von der Zieladresse **R0** zu der Bestimmungsadresse **R1** zu kopieren, und dabei die Reihenfolge aller Bytes umzukehren, kann man wie folgt vorgehen.

Assembler

#100

```

1 copy_memory_rev:        ; R0: source pointer s
2                         ; R1: destination pointer d
3                         ; R2: number of bytes N, N > 0
4 ADD       R1, R1, R2     ; d += N
5 next:                 ; R3 ← *s++
6 LDRB      R3, [R0], #1   ; *--d ← R3
7 STRB      R3, [R1], #-1!  ; N ← N - 1, set flags
8 SUBS      R2, R2, #1     ; continue, if N ≠ 0
9 BNE       next

```

Wenn man darauf achtet, dass beide Adressen an Wort-Grenzen ausgerichtet sind, so kann man zum Kopieren eines Speicherblockes folgende Implementierung verwenden.

Assembler

#101

```

1 copy_memory_aligned:    ; R0: source pointer s
2                         ; R1: destination pointer d
3                         ; R2: number of bytes N, N ≥ 4
4 MOV       R4, R2, LSR #2 ; R4: number of words k = N ÷ 4 to be copied
5                         ; N mod 4 bytes remaining
6 next:                 ; R3 ← *s++ (word)
7 LDR      R3, [R0], #4    ; *d++ ← R3 (word)
8 STR      R3, [R1], #4    ; k ← k - 1, set flags
9 SUBS     R4, R4, #1      ; continue, if k ≠ 0
10 BNE      next          ; half word left to copy?
11 TST      R2, #2          ; byte left to copy?
12 LDRNEH   R3, [R0], #2
13 STRNEH   R3, [R1], #2
14 TST      R2, #1          ; byte left to copy?
15 LDRNEB   R3, [R0]
16 STRNEB   R3, [R1]

```

Die Anzahl der Schleifendurchläufe beträgt jetzt nur noch ein Viertel gegenüber der zuvor vorgestellten Lösung, weil pro Lade- und Schreibbefehl 4 Bytes übertragen werden. Auch die Anzahl der Speicherzugriffe hat sich entsprechend verringert (bei einem 32-Bit-Datenbus). Falls nach Übertragung der $\frac{N}{4}$ Wörter noch ein Halbwort und/oder ein Byte zu übertragen sind (maximal 3 Bytes), werden diese Kopieroperationen zum Schluss angehängt.

Im Internet findet sich ein seltsamer, vielleicht sogar genialer Hack zum Kopieren ganzer Datenblöcke via *loop unrolling*, das sog. *Duff's Device*, nach Tom Duff von Lucasfilm (erfunden am 9. November 1983), nachfolgend aus didaktischen Gründen leicht modifiziert am Beispiel von Daten des Typs `data_t` (hier `unsigned`).

```

1 copy:                                ; k ← numOfItems / 4
2 MOV      R3, R2, LSR #2
3
4 copy_block:                           ; kopiere 4 Werte
5 LDMIA   R0!, {R4-R7}                 ; 4 Werte in R4 bis R7 laden ...
6 STMIA   R1!, {R4-R7}                 ; und dann in den Zielbereich schreiben.
7 SUBS    R3, R3, #1                  ; k ← k - 1 und Flags setzen
8 BNE     copy_block                 ; nochmals 4 Werte kopieren, wenn k > 0
9
10 ANDS   R2, R2, #3                ; Anzahl der verbleibenden Werte bestimmen
11 BNE     done
12
13 copy_value:                         ; Anzahl der verbleibenden Werte bestimmen
14 LDR     R3, [R0], #4
15 STR     R3, [R1], #4
16 SUBS   R2, R1, #1
17 BNE     copy_value:
18
19 done:                               PC, LR
20

```

Wenn man die Blockgröße von 4 Wörtern auf 8 Wörter erhöht, lässt sich die Ausführungsgeschwindigkeit noch weiter steigern.

4.10.7.6 Länge einer Zeichenkette bestimmen

Ein ASCII-Z-String⁵⁰ hat eine variable Länge (Anzahl der Zeichen **ohne** das abschließende '\0'). Diese Länge muss ggf. zur Laufzeit bestimmt werden. In **R0** wird der Zeiger auf die Zeichenkette übergeben. Diese Zeichenkette wird nicht verändert. Die Länge der Zeichenkette wird in **R0** zurückgegeben. Die Register **R1** bis **R4** dürfen verändert werden.

```

1 string_length:                      ; R0: source pointer s
2 MOV      R1, R0                     ; R1 ← source pointer s
3 MOV      R0, #0                     ; R0: number of characters n ← 0
4 string_length_loop:                 ; R2: character c ← *s++
5 LDRB    R2, [R1], #1               ; check end of string
6 CMP     R2, #0                     ; n ← n + 1 if no end of string
7 ADDNE   R0, R0, #1
8 BNE     string_length_loop       ; load next character if no end of string
9 MOV      PC, LR                   ; return n on end of string

```

4.10.7.7 Kopieren einer Zeichenkette

Beim Kopieren eines ASCII-Z-String ist die Anzahl der zu kopierenden Bytes nicht vorab bekannt. Es muss zur Laufzeit das String-Ende geprüft werden. In **R0** wird der Zeiger auf die Zeichenkette übergeben. Diese Zeichenkette wird nicht verändert. In **R1** wird der Zeiger auf den Zielpuffer übergeben. Die Register **R2** bis **R4** dürfen verändert werden.

```

1 string_copy:                        ; R0: source pointer s
2
3
4 LDRB    R3, [R0], #1               ; R1: destination pointer d
5 CMP     R3, #0                     ; R2: buffer length N, N > 0
6 BEQ     finish                    ; R3 ← *s+
7 STRB    R3, [R1], #1               ; prüfe ob R3 = 0
8 SUBS   R2, R2, #1                 ; String beenden wenn R3 = 0
9 BGT    string_copy              ; R3 ← *s++ ← R3
10 MOV    R3, #0                     ; N ← N - 1 und prüfen ob größer 0
11
11 finish:                           ; weiter mit nächstem Byte, wenn N > 0
12 STRB    R3, [R0]                  ; 0 (String-Ende) laden
13 MOV    PC, LR                   ; *d ← 0

```

4.10.7.8 Zeichen in einem String zählen

Es soll überprüft werden, wie oft in einem ASCII-Z-String der Buchstabe 'a' enthalten ist. In **R0** wird der Zeiger auf die Zeichenkette übergeben. Die Anzahl *n* der Buchstaben 'a' wird in **R0** zurückgegeben.

```

1 count_a:                            ; R0: pointer to string
2 STM     SP!, {R1-R3, LR}
3 MOV     R1, R0                     ; R1 ← pointer to string
4 MOV     R0, #0                     ; R0: number n: R0 ← 0
5 count_a_check_char:                ; R3: character c ← *string++
6 LDRB    R3, [R1], #1               ; check character
7 CMP     R3, #61
8 ADDEO   R0, R0, #1
9 BEQ     count_a_check_char      ; n ← n + 1, if c = 'a'
10 CMP    R3, #0                     ; check end of string
11 BNE     count_a_check_char      ; repeat if no end of string
12 LDM     SP!, {R1-R3, PC}          ; return n

```

Instruktion	besondere Bedingung	Zyklen		
		N	S	I
jede	Condition-Code nicht erfüllt		1	
Datenmanipulation	$Rd \neq PC$, ohne Barrel-Shifter		1	
Datenmanipulation	$Rd \neq PC$, mit Barrel-Shifter, vgl. $\langle S_{01} \rangle, \langle S_{02} \rangle$		1	1
Datenmanipulation	$Rd = PC$, ohne Barrel-Shifter	1	2	
Datenmanipulation	$Rd = PC$, mit Barrel-Shifter, vgl. $\langle S_{01} \rangle, \langle S_{02} \rangle$	1	2	1
MUL		1	m	
MLA		1	$m + 1$	
UMULL, SMULL		1	$m + 1$	
UMLAL, SMLAL		1	$m + 2$	
B, BL		1	2	
LDR	$Rd \neq PC$, vgl. $\langle S_{12} \rangle$	1	1	1
LDR	$Rd = PC$, vgl. $\langle S_{12} \rangle$	2	2	1
STR		2		
LDM	$PC \notin \text{Liste}$, vgl. $\langle S_{17} \rangle$	2	$n - 1$	1
LDM	$PC \in \text{Liste}$, vgl. $\langle S_{17} \rangle$	2	$n + 1$	1
STM		2	$n - 1$	
SWP		2	1	1
MSR, MRS			1	
SWI		1	2	

Tab. 40: Anzahl und Art der Zyklen zur Ausführung unterschiedlicher ARM7TDMI-Befehle mit ggf. unterschiedlichen Randbedingungen.