



6. Cortex-M4-Prozessor – Aufbau und Adressierungsarten –

6.1 Gegenstand der Vorlesung

- Interne Organisation
 - Architektur
 - Register
 - Statusbits
 - Speicherstruktur
- Basis-Adressierungsarten
 - direct
 - absolute
 - immediate
 - indirect





6.2 Grobe Übersicht der Entwicklungsgeschichte und Einsatzbereiche

Prozessor	Einf.	Takt. (MHz)	Besonderheiten	Anwendungen
ARM 2	1986	8	4 MIPS, + Multiply-Unit	
ARM 3	1989	25	12 MIPS + 4k Cache	
ARM 6		33	28 MIPS, ext. FPU	Psion-PDA
ARM 7 TDMI-S	1993	bis 72	36MIPS bei 40 MHz	Game-Boy, iPod, Lego-NXT, versch. PDA's , Navi's und Handy's, ...
Strong ARM	1996	233	16k /16k Cache, MMU	Apple-Newton, iPAQ-PDA's, Palm, ...
ARM 9E	1997	200	220MIPS bei 200 Mhz	Handy's: Sony Ericsson, Siemens, BenQ,
XScale (z.B. PXA27x)	2002	ca. 300 - 600	800 MIPS bei 624 Mhz, 32k /32k Cache, MMU	Palm Tx, div. PDA's (Dell, HP, Siemens, Toshiba, ..)
Cortex - Serie	2002.		Thumb-2-Befehlssatz	s.u.



Cortex-A-Serie

Zweck: High-Performance, low Power, für offene Betriebssysteme (z.B. Android)

Anwnd.: Smartphones, Reader, Netbooks,

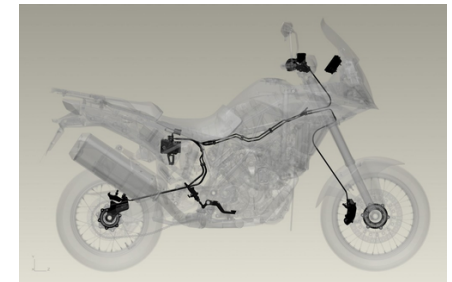
Beisp.: Samsung Galaxy SII, Kindle 2, Blackberry Playbook,



Cortex-R-Serie

Zweck: High-Performance u. Realtime-Systeme, Embedded-Anwendg.

Anwnd.: Airbag, Brems- und Steuersysteme, HD-Controller
Blue-Ray-Player, Medizinische Geräte, Modems ...



Cortex-M-Serie

Zweck: General-Purpose-Microcontroller, Mixed-Signal (A/D) ,
Embedded-Anwendg., kostengünstig, energieeffizient

Anwnd.: Küchengeräte, Taschenrechner, Anlagensteuerungen, ...





6.3 Cortex-M4-Architektur in Stichworten

- RISC (Reduced Instruction Set Computer) – Architektur
 - relativ wenige aber mächtige Befehle
 - viele Register (16)
 - 32 bit Adressbus / 32 bit Datenbus
-
- arithmetische/logische Operationen nur in Verbindung mit Registern möglich
 - Operanden müssen zuvor in die Register geladen werden (**ldr**-Befehl)
 - Ergebnisse müssen nach Berechnungsende in den Speicher geschrieben werden (**str**-Befehl)
 - besitzt keine direkte Adressierung
-
- Thumb-2-Befehlssatz (16/32-Bit-Befehle)
→ schneller und kompakter Code



Jede Instruktion belegt höchstens 16/32 bit

- Vorteil: schnelle Ausführung (Pipelining)

- daraus ergeben sich folgende Eigenschaften:

- Nur Konstanten $0 \dots 2^{16}$ können direkt angegeben werden und Linksverschiebungen von $0 \dots 255$ um $0 \dots 31$ Bitstellen. Andere Konstanten müssen etwas umständlich im Speicher abgelegt werden (s.u.).
- Programmverzweigungen nur möglich im Bereich PC +/- 32MByte. Größere Sprünge nur durch Umladen des PC.
- Direkte Adressierung von Speicher (mit LDR, STR) gibt es beim Cortex gar nicht, per Assembler-Pseudobefehl kann aber Bereich PC -255 + 4095 direkt adressiert werden. Für Fernzugriffe muss immer indirekt adressiert werden.



6.4 Speicherformate des ARM-Cortex-M4

Byte-Maschine : Jedes Byte hat eine eigene Adresse.

Datentypen : Byte, Halbwort (2 Byte), Wort (4 Byte).

Little Endian : Beide Modi (big endian / little endian) sind möglich.
Üblicherweise wird der ARM im **little endian** mode betrieben.
→ Das LSB (*least significant byte*) eines Halbwortes / Wortes liegt auf der kleinsten Adresse.

Aligned : Halbwort (2 Byte) müssen auf Halbwortgrenzen (= durch 2 teilbare Adressen) beginnen.
Worte (4 Byte) müssen auf Wortgrenzen (= durch 4 teilbare Adressen) beginnen.
Außnahmen möglich (LDR, STR), aber meist langsamer.



6.5 Register des ARM-Cortex-M4

6.5.1 Die Register *r0* – *r15*

16 x Register *r0*...*r15* (alle 32 bit)
im Standardmodus für Anwendungsprogramme (*User mode*)

r0...*r12* : frei verwendbar (bedingt *r11*, *r12*)

r13 : *Stackpointer* (SP)

→ per Konvention festgelegt

→ z.B. für die Parameterübergabe bei Unterprogrammaufrufen

r14 : *Link Register* (LR)

→ speichert die Rücksprungadresse bei Unterprogrammaufrufen

r15 : *Program Counter* (PC)

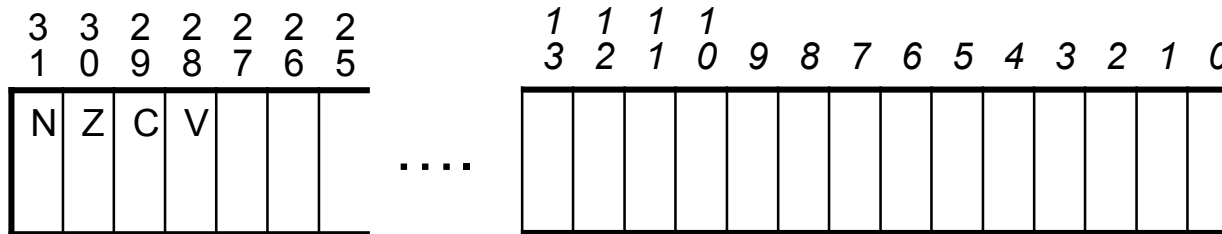
→ Bits 0 und 1 undefiniert (wg. 32-bit-Alignment)

→ enthält die Adresse des aktuell ausgeführten Befehls

→ wird nach jedem Befehl um 4 erhöht



6.5.2 Statusregister



N (Negative-Flag) : enthält das MSB eines Ergebnisses, d.h. 0=pos. 1=neg.

Z (Zero-Flag) : wird gesetzt, wenn das Ergebnis 0 ist.

C (Carry-Flag) : Übertrag arith. Operationen

V (Overflow-Flag) : zeigt einen Overflow bei signed-integer-Operationen an.

Alle anderen Flags spielen in dieser Vorlesung/Praktikum keine Rolle.

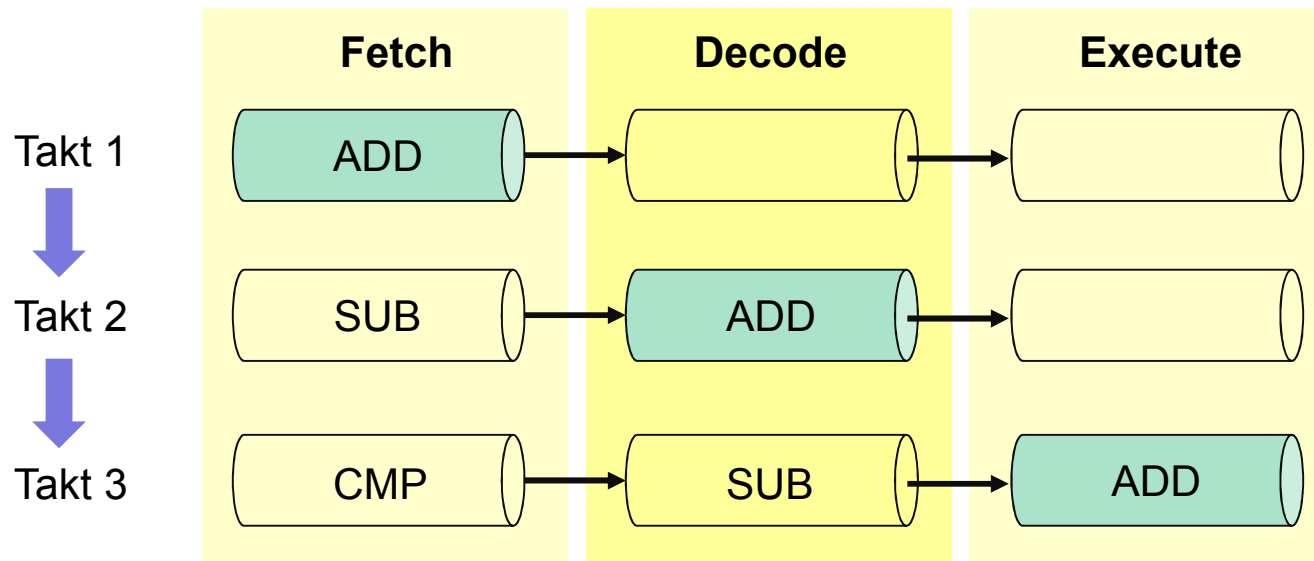


6.6 Befehlspipeline

Die Pipeline erhöht die Geschwindigkeit des Prozessors, indem mehrere aufeinanderfolgende Befehle ähnlich einer Fließbandproduktion abgearbeitet werden.

Der ARM-Cortex-M4 hat eine 3-stufige Befehlspipeline, mit den Stufen:

- | | | |
|-------------------|------------|-----------------------|
| 1. Fetch | Befehl n+2 | --> Befehl lesen |
| 2. Decode | Befehl n+1 | --> Befehl decodieren |
| 3. Execute | Befehl n | --> Befehl ausführen |





6.7 Grundsätzliches zum Befehlsaufbau und zur Notation

6.7.1 Anmerkung zur verwendeten Assemblernotation

Für die mnemonische Beschreibung von ARM-Maschinenprogrammen gibt es verschiedene Notationen, die sich zum Teil erheblich voneinander unterscheiden.

Im Rahmen dieser Vorlesung wird die Notation der Keil-Entwicklungsumgebung (μ Vision) verwendet !!



6.7.2 Befehlsaufbau bei einfachen Befehlen

Befehlsaufbau bei 2-Operanden-Befehlen:

Syntax: **Operator** Ziel, Quelle

Beispiel: `mov r5, r4 ; [r5] ← [r4]`

Befehlsaufbau bei 3-Operanden-Befehlen:

Syntax: **Operator** Ziel, Operator_1, Operator_2

Beispiel: `add r5, r4, r3 ; [r5] ← [r4]+[r3]`



6.7.3 Angabe von Konstanten

Binäre numerische Konstanten

Syntax $2_ \{ \text{binäre Ziffer} \}$

Beispiel: `mov r0, #2_01000001` `[r0] ← 65 (=0x41)`

Dezimale numerische Konstanten

Syntax `DezimalzifferOhneNull { Dezimalziffer }`

Beispiel: `mov r0, #65` `[r0] ← 65 (=0x41)`

Hexadezimale numerische Konstanten

Syntax `0x { hexadezimale Ziffer }`

Beispiel: `mov r0, #0x41` `[r0] ← 65 (=0x41)`

Zeichenkonstanten

Syntax `'Zeichen '`

Beispiel: `mov r0, #'A'` `[r0] ← 65 (=0x41)`



ÜBUNG: Angabe von Konstanten

Wie könnte man die Befehle noch schreiben ?

Welche Schreibweise ist am zweckmäßigsten (am besten lesbar).

- a) `mov r1, #41` ; Alter des Studenten
- b) `mov r2, #0x44` ; Lade Steuerbits
- c) `mov r3, #2_10000111` ; Lade Steuerbits
- d) `mov r4, #'C'` ; Lade den zu suchenden Buchstaben
- e) `mov r5, # -5` ; Lade Kontostand
- f) `mov r6, #65` ; Lade Alter des Großvaters



6.7.4 Bedingte Ausführung von Befehlen

Die Befehle des ARM können so angegeben werden, dass sie nur dann ausgeführt werden, wenn eine vorgebbare Bedingung (Flags N, Z, C, V) erfüllt ist.

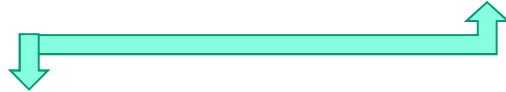
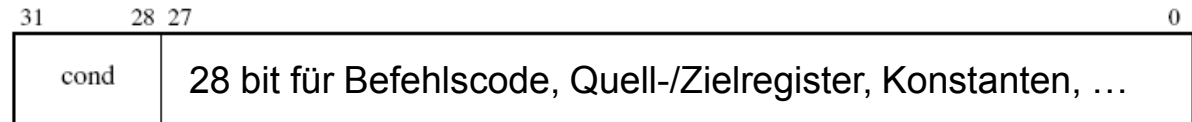
Auswahl einiger Bedingungen	MI	minus	N=1
	PL	plus	N=0
	EQ	equal	Z=1
	NE	not equal	Z=0
	CS	carry set	C=1
	CC	carry clear	C=0
	VS	overflow set	V=1
	VC	overflow clear	V=0

Beispiele:

```
moveq    r0, #100
addcs    r0, r1, #1
```

```
nur wenn Z=1 dann    [r0] ← 100
nur wenn C=1 dann    [r0] ← [r1] + 1
```

Insgesamt gibt es 15 unterschiedliche Bedingungen.

**Befehlskodierung :**

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
....			
....			
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-



6.7.5 Setzen der Flags in Abhängigkeit des Ergebnisses

Die arithmetischen/logischen Befehle des M4 können die Flags (N, Z, C, V) passend zum Ergebnis auf 0 bzw. 1 setzen.

Sofern dies gewünscht ist, wird dem Befehl ein **S** angehängt.

Handelt es sich um einen bedingten Befehl, so werden die Flags nur dann verändert, wenn der Befehl auch ausgeführt wird.

Beispielprogramm:

1	mov	r1, #1	[r1] ← 1
2	mov	r2, #2	[r2] ← 2
3	sub s	r0, r1, r2	[r0] ← [r1] - [r2] N=1, C=0, V=0, Z=0
4	add pls	r0, r0, #5	[r0] ← [r0] + 5 <u>wird nicht ausgeführt</u> da N=1 ist (das Ergebnis von 3 ist negativ). Trotz angehängtem S werden die Flags daher auch nicht verändert.



Beispiel: Programmierung einer einfachen Entscheidung

```
if (Z = 1) then  
    r0 ← r0 + 1  
else  
    r0 ← r0 - 1
```

addeq	r0, #1	wenn Z=1 dann $r0 \leftarrow r0 + 1$
subne	r0, #1	wenn Z=0 dann $r0 \leftarrow r0 - 1$

Anmerkungen zum Beispiel:

- Wichtig: nicht **addeqs** verwenden ! (könnte dazu führen, dass nach dem if-Block auch der else-Block durchgeführt wird)
- Nur für wenig geschachtelte Alternativen effizient.





6.8 Einfache Speicherzugriffe

6.8.1 Load-store-Architektur

Bevor mit Speicherinhalten gerechnet werden kann, müssen die Speicherinhalte immer erst in die Register geladen werden.

Eine typische Verarbeitungssequenz ist beim ARM also wie folgt aufgebaut:

1. Speicherwerte in das/die Register laden. → **ldr**-Befehl (*load register*)

2. Berechnungen in den Registern ausführen

3. Ergebnis wieder in den Speicher schreiben. → **str**-Befehl (*store Register*)



6.8.2 LDR-Befehl: Laden von Speicherinhalten in Register

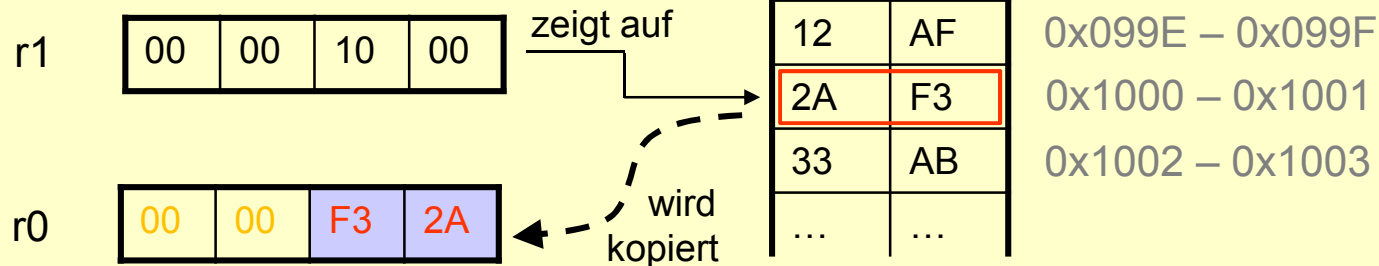
Aufruf: ldr Zielregister, [Adressregister]

Quelladressierung : indirect

Arbeitsweise:

- [Zielregister] \leftarrow [M([Adressregister])]
In Worten: Im Adressregister steht die Zugriffsadresse. Das auf der Zugriffsadresse stehende Datenwort wird in das Zielregister geladen.
- Mit **ldrb** und **ldrh** können auch Bytes oder Halbworte geladen werden. Dabei werden die vorderen (nichtbenötigten) Bits mit 0 besetzt.
- Bei Halbwort- und Wortzugriffen ist evtl. das Alignment zu beachten (prozessorabh.)

Beispiel: ldrh r0, [r1]





6.8.3 STR-Befehl: Schreiben von Registerinhalten in den Speicher

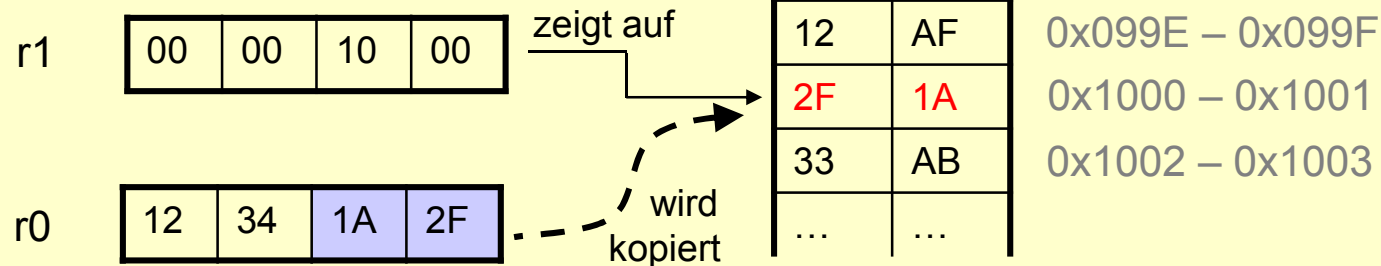
Aufruf: `str Quellregister, [Adressregister]`

Quelladressierung : *indirect*

Arbeitsweise und Besonderheiten:

- $[M([\text{Adressregister}])] \leftarrow [\text{Quellregister}]$
In Worten: Im Adressregister steht die Zieladresse. Das im Quellregister stehende Datenwort wird auf die Zieladresse geschrieben.
- Mit **strb** und **strh** können auch Bytes oder Halbworte geschrieben werden.
- Bei Halbwort- und Wortzugriffen ist evtl. das Alignment zu beachten (prozessorabh.)

Beispiel: `strh r0, [r1]`





6.9 Einfache Registeroperationen

6.9.1 Übersicht

Der ARM-Befehlssatz umfasst relativ wenige Grundoperationen (RISC), wie:

- Konstanten oder Registerinhalte kopieren
- Mathematische Operationen
 - Addieren
 - Subtrahieren
 - Vergleichen
 - Multiplizieren
- Logische Operationen
 - bitweises AND
 - bitweises OR
 - bitweises XOR

Diese Befehle werden sehr effizient ausgeführt. Komplexere Operationen müssen aus diesen Befehlen zusammengesetzt werden.



6.9.2 MOV-Befehl: Konstanten in Register schreiben

```
mov  Zielregister,  #Konstante
```

Quelladressierung : *immediate*

Besonderheiten:

- Der Cortex-M4 erlaubt beliebige 16-Konstanten (unsigned) 0 65535
- oder Konstanten 0...255 linksverschoben um 0 31
- Durch mov werden immer alle 32-bit des Zielregisters verändert.

Beispiele:

- mov r0, #0xff [r0(0:31)] ← 255
- mov r0, #0xfffe [r0(0:31)] ← 65534
- mov r0, #0x10005 geht nicht !
- mov r0, #0xff000000 geht (255 << 24)

```
mov  r0, #255
```

vorher

r0

12	34	58	9A
----	----	----	----

nachher

00	00	00	ff
----	----	----	----



ÜBUNG: Adressierungsart immediate - erlaubte Konstanten

Welche Konstanten sind bei der Adressierungsart „*immediate*“ erlaubt ?

- a) 234
- b) 2357
- c) 0xfe02
- d) 0xfe0000
- e) 0xfe0002
- f) 'A'



6.9.3 MOV-Befehl: Registerinhalte in andere Register kopieren

mov Zielregister, Quellregister

Quelladressierung : *register*

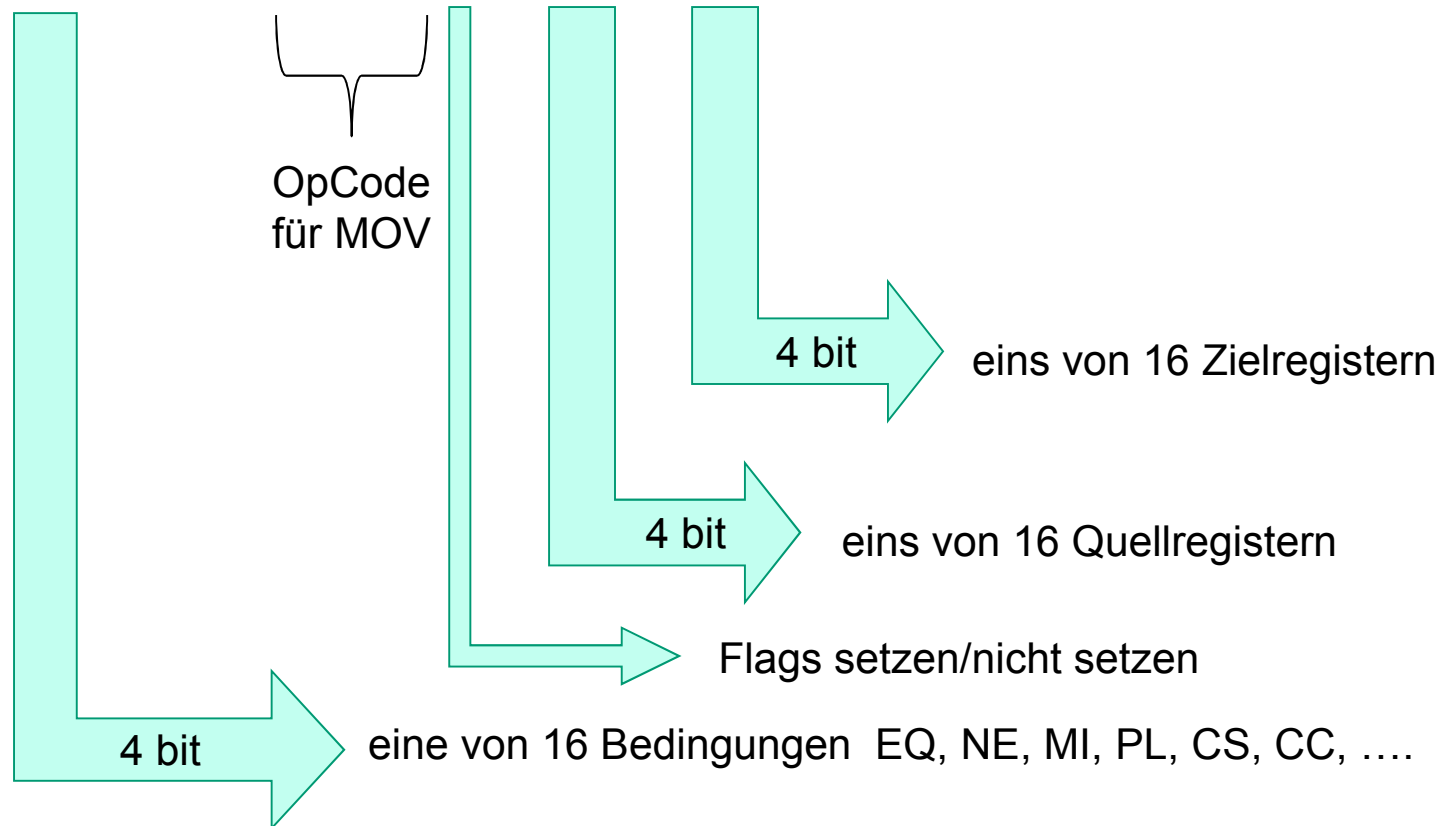
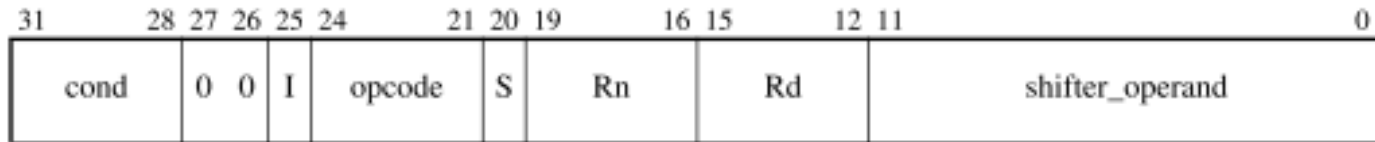
Beispiele:

- mov r0, r1 $[r0] \leftarrow [r1]$
- moveq r0, pc wenn Z=1 $[r0] \leftarrow [pc]$ Anm.: pc = r15
- movnes r5, r9 wenn Z=0 $[r5] \leftarrow [r9]$, Flags ggf. modifizieren



Befehlskodierung :

MOV {<cond>}{S} <Rd>, <Rn>





6.9.4 Einfache logische/arithmetische Operationen

Aufruf: Operation Zielregister, {Quellregister} , # Konstante
 Operation Zielregister, Quellregister1 , Quellregister2

Operationen: AND, EOR, SUB, ADD, ORR, ... (insgesamt 16)

Besonderheiten:

- ADD, SUB: beliebige 12-Konstanten (unsigned) 0 4095
- Alle: Konstanten 0...255 linksverschoben um 0 31

Beispiele:

• add	r0, #1	[r0] ← [r0] +1	
• sub	r0, r1, #2	[r0] ← [r1] - 2	
• and	r0, #0xf0	[r0] ← [r0] AND 2_11110000	
• ands	r0, r1, #2	[r0] ← [r1] AND 0x2	ggf. N- und Z-Flag ändern
• andeq	r0, r1, #2	[r0] ← [r1] AND 0x2,	wenn Z=1 ist
• orr	r0, r1, r2	[r0] ← [r1] OR [r2]	



ÜBUNG: Adressierungsarten "register, immediate, indirect"

Geben Sie jeweils die Registerinhalte bzw. Speicherinhalte nach den Befehlen an. Alle verwendeten Register sind mit **0xFFFFFFFF** initialisiert. Im Speicher stehen folgende Werte (alles Hexadezimal):

Adr.	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
Inh.	12	23	34	45	56	67	78	89	9A	AB

			3 +0	2 +1	1 +2	0 +3
mov	r0, #0xaa	[r0] =				
mov	r1, #170	[r1] =				
movs	r2, #0xaa00	[r2] =				
moveq	r3, #0x0aa000	[r3] =				
mov	r4, #0x1000	[r4] =				
ldrb	r5, [r4]	[r5] =				
ldr	r6, [r4]	[r6] =				
add	r4, #4	[r4] =				
str	r6, [r4]	M[0x1004] =				
add	r4, #4	[r4] =				
strh	r6, [r4]	M[0x1008] =				



6.10 Erweiterte Speicherzugriffe

6.10.1 Hintergrund

In den meisten Programmen werden folgende Operationen durchgeführt

- Bearbeitung von Strings
- Zugriff auf Felder (einfache/komplexe Datentypen)
- Unterprogrammsprünge

Für die effiziente Durchführung dieser Operationen besitzt der ARM erweiterte Adressierungsarten.



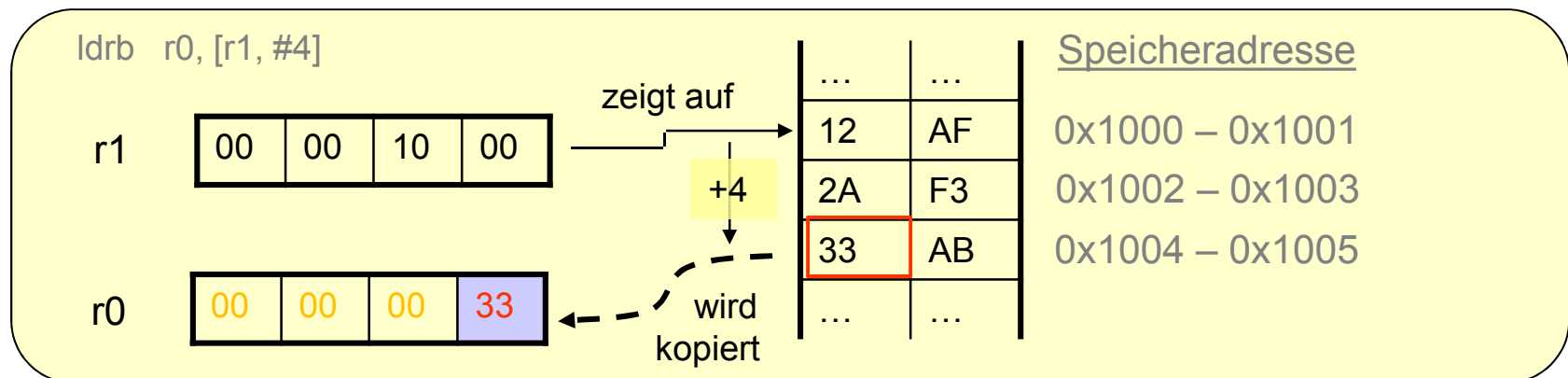
6.10.2 Speicher laden mit konstantem Offset zu einer Basisadresse

Aufruf: `ldr Zielregister, [Basisadressregister, #Offset]`

Quelladressierung : *preindex (immediate offset)*

Arbeitsweise und Besonderheiten:

- `[Zielregister] ← [M([Basisadressregister] + Offset)]`
- **ldr** lädt Wort, **ldrb** lädt Byte (die vorderen 3 Byte des Registers werden mit 0 besetzt)
- Der Offset muss im Bereich -255 ... +4095 liegen.
- Nützlich bei Zugriff auf **Tabellen, Unterprogrammparameter, Datenfelder**
- **str** speichert Wort, **strb** speichert ein Byte





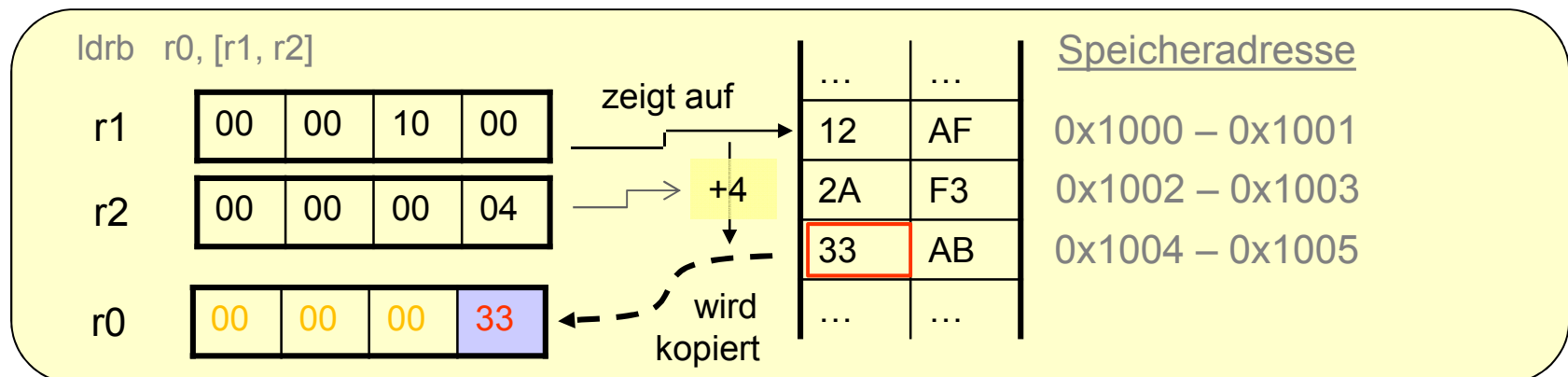
6.10.3 Speicher laden mit variablem Offset zu einer Basisadresse

Aufruf: `ldr Zielregister, [Basisadressregister, Offsetregister]`

Quelladressierung : *preindex (register offset)*

Arbeitsweise und Besonderheiten:

- $[Zielregister] \leftarrow [M([Basisadressregister] + [Offsetregister])]$
- **ldr** lädt Wort, **ldrb** lädt Byte (die vorderen 3 Byte des Registers werden mit 0 besetzt).
- Nützlich bei Zugriff auf **Tabellen und Strings**.
- **str** speichert Wort, **strb** speichert ein Byte





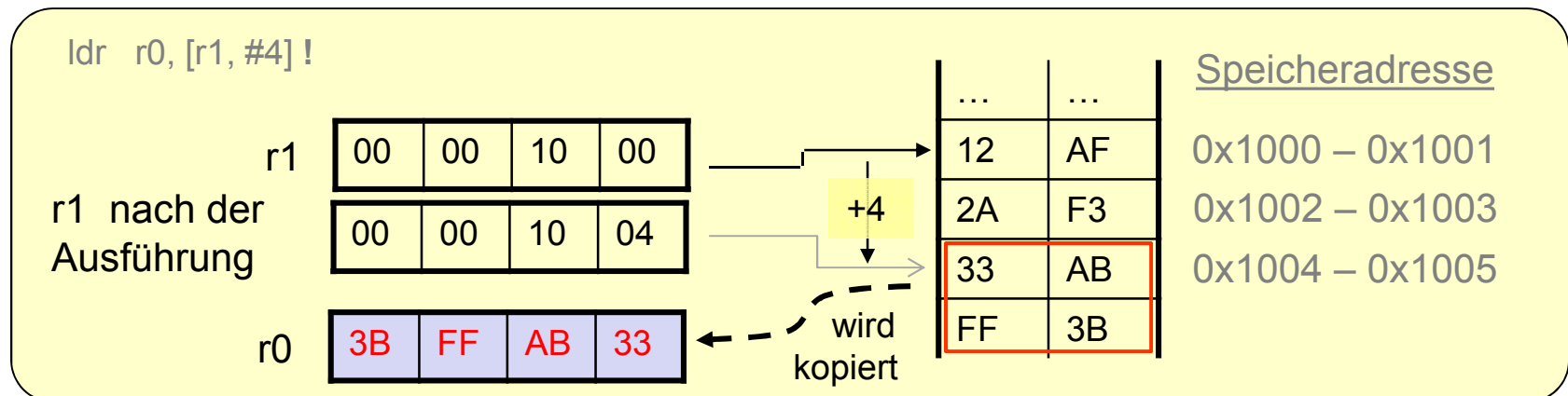
6.10.4 Speicher laden mit konstantem Offset, pre-indexing und update

Aufruf: `ldr Zielregister, [Basisadressregister, # Offset] !`

Quelladressierung : *preindex (immediate offset) with writeback*

Arbeitsweise und Besonderheiten:

- 1. `[Zielregister] ← [M([Basisadressregister] + Offset)]`
- 2. `[Basisadressregister] ← [Basisadressregister] + Offset`
- **ldr** lädt Wort, **ldrb** lädt Byte (die vorderen 3 Byte des Registers werden mit 0 besetzt)
- Der Offset muss im Bereich `-255 ... +255` liegen.
- Nützlich beim Abarbeiten von **Tabellen/Zeichenketten** und bei **Stackoperationen**
- **str** speichert Wort, **strb** speichert ein Byte





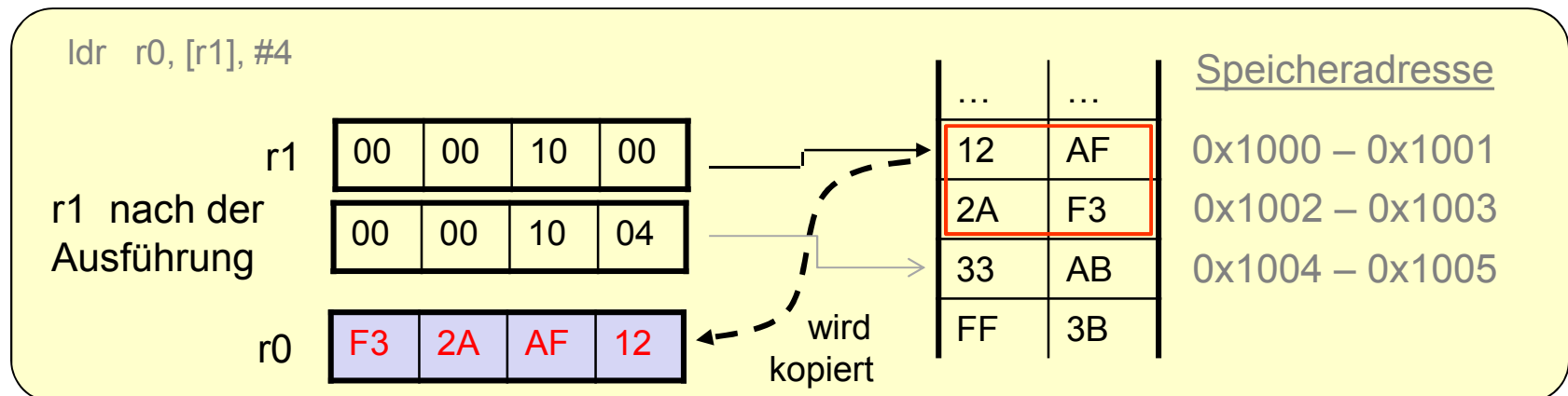
6.10.5 Speicher laden mit konstantem Offset, post-indexing und update

Aufruf: `ldr Zielregister, [Basisadressregister] , # Offset`

Quelladressierung : *postindex (immediate offset)*

Arbeitsweise und Besonderheiten:

- 1. `[Zielregister] ← [M([Basisadressregister])]`
- 2. `[Basisadressregister] ← [Basisadressregister] + Offset`
- **ldr** lädt Wort, **ldrb** lädt Byte (die vorderen 3 Byte des Registers werden mit 0 besetzt)
- Der Offset muss im Bereich -255 ... 255 liegen.
- Nützlich beim Abarbeiten von **Tabellen/Zeichenketten** und bei **Stackoperationen**
- **str** speichert Wort, **strb** speichert ein Byte





ÜBUNG: Adressierungsarten „immediate offset, register offset, pre-indexed, post-indexed“

Geben Sie jeweils die Registerinhalte bzw. Speicherinhalte nach den Befehlen an.

Alle verwendeten Register sind mit **0xFFFFFFFF** initialisiert.

Im Speicher stehen folgende Werte (Hex.):

Adr.	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B
Inh.	10	06	34	45	56	67	78	89	9A	AB	BC	CD

			3 +0	2 +1	1 +2	0 +3
mov	r0, #0x1000	[r0] =				
ldr	r1, [r0, #4]	[r1] =				
strb	r1, [r0, #2]	[M(0x1000)] =				
ldr	r2, [r0, #4]!	[r2] =				
ldr	r3, [r0, #4]!	[r3] =				
ldrb	r4, [r0], #1	[r4] =				
ldrb	r5, [r0], #1	[r5] =				
ldrb	r6, [r0], #1	[r6] =				
str	r1, [r0, #-8]	[M(0x1002)] =				



6.11 Erweiterte Registeroperationen

6.11.1 Hintergrund

In vielen Programmen kommen folgende Operationen vor:

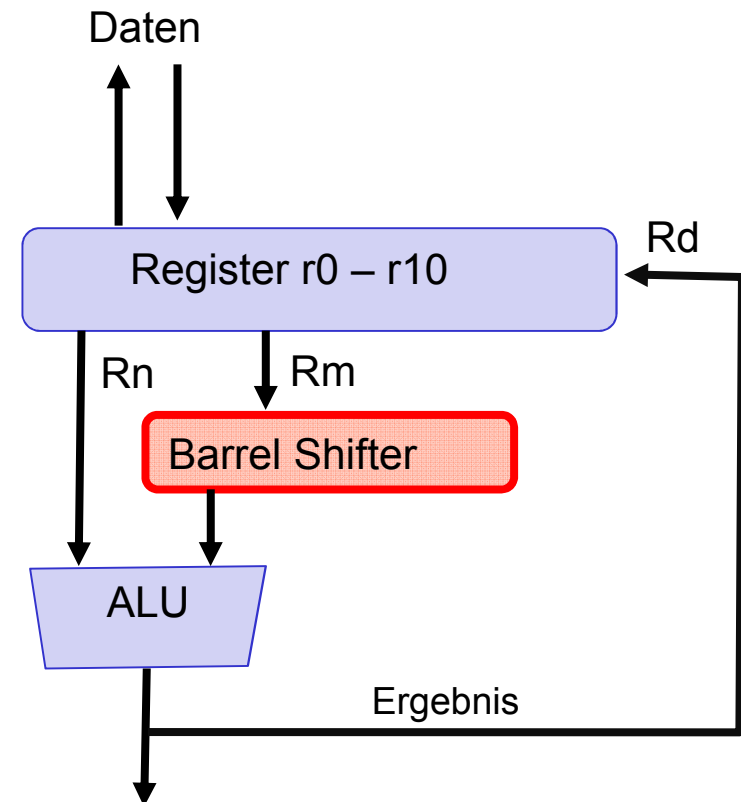
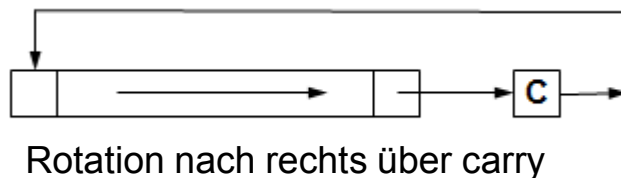
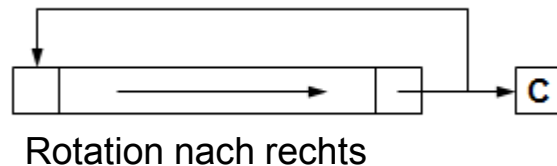
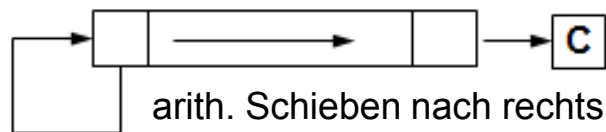
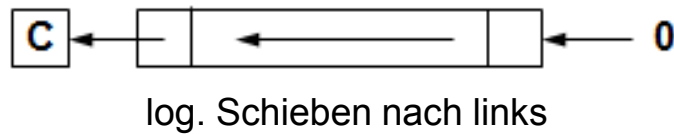
- indizierte Zugriffe auf Halbwort- und Wortfelder
- schnelle Multiplikationen mit 2^n ,
- Division,
- Bitmanipulationen,
- mathematische Operationen (Quadratwurzel, Logarithmus, trigonometr. Fkt.),
- Erzeugung von Zufallszahlen.

Für die effiziente Durchführung dieser Operationen besitzt der ARM einen sog. **Barrel-Shifter**.



6.11.2 Barrel Shifter

- Erlaubt Bit-Verschiebungen und –Rotationen um einen oder mehrere Bitpositionen in einem Schritt.
- Typ. Operationen sind:





6.11.3 Shifted-Register-Operand

Bei den meisten Registeroperationen kann der letzte Operand vor der Operation um bis zu 32 bit verschoben oder rotiert werden.

Fünf Typen von Schiebeoperationen stehen zur Verfügung:

ASR	Arithmetisches Schieben nach rechts
LSL	Logisches Schieben nach links
LSR	Logisches Schieben nach rechts
ROR	Rotieren nach rechts
RRX	Rotieren nach rechts über carry

Beispiele:

- mov r2, r0, **LSL #2** [r2] ← LSL2(r0) d.h. [r0] * 4
- add r9, r5, r5, **LSL #2** [r9] ← [r5] + LSL2(r5) d.h. [r5] * 5
- sub r0, r4, r5, **LSL #2** [r0] ← [r4] - LSL2(r5) d.h. [r4] - [r5]*4
- sub r0, r4, r5, **LSR #3** [r0] ← [r4] - LSR2(r5) d.h. [r4] - [r5]/8
- mov r2, r0, **LSL r3** [r2] ← [r0] linksverschoben um den Wert in r3



6.11.4 Befehlskodierung

