



## 8. ARM-Cortex-Befehle im Detail

- Ganzzahlarithmetik
- Kontrollstrukturen
- Bitmanipulation
- Schieben und Rotieren



## 8.1 Zur Notation des letzten Operators (N)

Bei den Registeroperationen (mov, add, sub, and, ....) hat der letzte Operator eine der folgenden Formen:

---

Operation N	Syntax des letzten Operanden
-------------	------------------------------

---

Immediate	<i># imm</i>
Register	<i>Rm</i>
Log. shift left by immediate	<i>Rm, LSL # shift</i>
Log. shift left by register	<i>Rm, LSL Rs</i>
Log. shift right by immediate	<i>Rm, LSR # shift</i>
Log. shift right by register	<i>Rm, LSR Rs</i>
Arith. shift right by immediate	<i>Rm, ASR # shift</i>
Arith. shift right by register	<i>Rm, ASR Rs</i>
Rotate right by immediate	<i>Rm, ROR # shift</i>
Rotate right by register	<i>Rm, ROR Rs</i>
Rotate right with extend	<i>Rm, RRX</i>



## 8.2 Ganzzahl-Arithmetik

### 8.2.1 Übersicht

<b>add</b>	Binäre Addition
<b>adc</b>	Binäre Addition mit Berücksichtigung von Überträgen
<b>sub</b>	Binäre Subtraktion
<b>sbc</b>	Binäre Subtraktion mit Berücksichtigung von Überträgen
<b>rsb</b>	Umgekehrte binäre Subtr.
<b>rsc</b>	Umgekehrte binäre Subtr. mit Berücksichtigung von Überträgen
<b>cmp</b>	Vergleich zweier Integer-Operanden (verändert nur die Flags)
<b>cmn</b>	Vergleiche negiert (verändert nur die Flags)
<b>neg</b>	Negieren gemäß 2-er-Komplement
<b>mul</b>	Multiplikation zweier Zahlen (verschiedene Varianten)



## 8.2.2 Addition und Subtraktion

**Aufruf:** <instruction>{<cond>}{S} Rd, Rn, N

<b>ADD</b>	Addieren zweier 32-bit Werte	$Rd = Rn + N$
<b>ADC</b>	Addieren zweier 32-bit Werte mit carry	$Rd = Rn + N + carry$
<b>SUB</b>	Subtraktion zweier 32-bit Werte	$Rd = Rn - N$
<b>SBC</b>	Subtraktion zweier 32-bit Werte mit carry	$Rd = Rn - N - \sim carry$
<b>RSB</b>	inverse Subtraktion zweier 32-bit Werte	$Rd = N - Rn$
<b>RSC</b>	inverse Subtraktion zweier 32-bit Werte mit carry	$Rd = N - Rn - \sim carry$

### Beispiele:

- **add**     r0, r1                     $[r0] \leftarrow [r0] + [r1]$
- **subeq**   r0, r1, #2                 $[r0] \leftarrow [r1] - 2, \text{ falls } Z=1$
- **subs**    r0, r1, r2, LSL #2        $[r0] \leftarrow [r1] - [r2]*4, \text{ Flags passend setzen}$



### 8.2.3 Vergleich

**Aufruf:** <instruction>{<cond>} Rn, N

Bei den Vergleichsoperationen werden **nur die Flags** verändert, d.h. es gibt **kein Zielregister** !

**CMN**      Vergleiche negiert       $Rn + N$

**CMP**      Vergleiche       $Rn - N$

**TEQ**      Vergleiche auf Gleichheit zweier 32-bit Werte       $Rn \text{ xor } N$   
→ Z=1 , wenn  $Rn = N$

**TST**      Teste Bits 32-bit Werte       $Rn \text{ and } N$   
→ Z=1 , wenn es keine 1-Übereinstimmungen gibt

#### Beispiele:

- cmp      r0, r1       $[r0] - [r1]$ , Flags entsprechend dem Ergebnis
- teq      r0, #0x11110000       $[r0] \text{ xor } 0xF0$ , Flags entsprechend dem Ergebnis
- tst      r0, #0xf0       $[r0] \text{ and } 0xF0$ , Flags entsprechend dem Ergebnis



## ÜBUNG: Vergleiche

Wie sind die Flags N, Z, C, V nach folgenden Befehlen gesetzt?

```
mov r0, #15
mov r1, #30
mov r2, #-34          ; wird ersetzt durch: mvn r2, #33
mov r3, #2_10100001   ; 0xA1 = 161
```

```
cmp r0, r1
cmp r1, r0
cmp r1, r0, LSL #1
```

```
teq r3, #161
teq r3, #0xFF
```

```
tst r3, #2_01011110
tst r3, #2_00000001
tst r3, #0
```



## 8.2.4 Multiplikation von unsigned und signed Werten

**Aufruf:** mla {<cond>}{S} Rd, Rm, Rs, Rn  
mul {<cond>}{S} Rd, Rm, Rs

<b>MLA</b>	Multiplikation mit anschließender Addition	$Rd = Rm * Rs + Rn$
<b>MUL</b>	Multiplikation zweier 32-bit Werte	$Rd = Rm * Rs$

Achtung: Die Multiplikation zweier 32-Bit-Werte kann ein 64-Bit-Ergebnis zur Folge haben. Ein 32-Bit-Register reicht daher für das Ergebnis nicht aus.

Es gibt weitere Multiplikationsbefehle (*long multiply*) für signed und unsigned Integerzahlen. Dort wird das Ergebnis auf 2 Register (=64 Bit) aufgeteilt.



## ÜBUNG: Arithm. Ausdruck

Schreiben Sie ein Programm, welches den folgenden Ausdruck berechnet:

$$(X^2 + Y^2) - (X+Y)^3$$

mit  $X=+5$ ,  $Y=-7$ .

Worauf ist zu achten?





## 8.3 Kontrollstrukturen

### 8.3.1 Übersicht

Zweck von Kontrollstrukturen:

- Fortführung des Programms an anderer Stelle ohne weitere Bedingung = **unbedingter Sprung**
- Fortführung des Programms an anderer Stelle, wenn eine bestimmte Bedingung erfüllt ist = **bedingter Sprung**

Anwendungen:

- Schleifen (while, for, do...while)
- bedingte Ausführung (if ... then ... else, switch ... case)
- Unterprogrammsprünge



### 8.3.2 *Bedingter oder Unbedingter Sprung (Branch)*

#### Befehl:

b {<cond>} Label

#### Wirkung:

- Sofern die Bedingung <cond> erfüllt ist, wird die Programmausführung an der angegebenen Stelle (label) fortgesetzt.  
Der Programmzähler wird auf die Adresse des Sprungzieles gesetzt.
- Das Sprungziel (Label) muss in einem Bereich von **+/- 32MB** liegen, da es PC-relativ im Befehl codiert wird (→ short branch).

#### Beispiele:

```
b Lab1 ; springe in jedem Fall nach Lab1
.....
Lab1 mov r0, #22
```

```
beq Lab2 ; springe nach Lab1, wenn Z=1
.....
Lab2 mov r0, #22
```



### 8.3.3 Arten von Bedingungen

<b>&lt;cond&gt;</b>	<b>Bedeutung</b>	<b>Flags</b>
EQ	equal	$Z == 1$
NE	not equal	$Z == 0$
CS/HS	carry set (higher or same)	$C == 1$
CC/LO	carry clear (lower)	$C == 0$
MI	minus	$N == 1$
PL	plus	$N == 0$
VS	overflow set	$V == 1$
VC	overflow clear	$V == 0$
HI	higher	$C == 1 \ \&\& \ Z == 0$
LS	lower or same	$C == 0 \    \ Z == 1$
GE	greater than or equal	$N == V$
LT	less than	$!N == V$
GT	greater than	$N == V \ \&\& \ Z == 0$
LE	less than or equal	$!N == V \    \ Z == 1$



### 8.3.4 Anwendung bedingter Sprungbefehle

Sehr häufig (aber nicht zwingend) werden Sprungbefehle in Verbindung mit dem Compare-Befehl (= subs ohne Ergebnis) verwendet.

cmp	r0, r1	; Setzen der Condition-Codes gemäss des ; Ergebnisses der Operation [r0] – [r1]
bcc	Marke	; Verzweigt, wenn cc gilt

Sprungbefehle können aber auch in Verbindung mit anderen Befehlen (mit Befehlszusatz **s**) verwendet werden, sofern diese Auswirkungen auf die verwendeten Flags haben (Verschiebebefehle, log. Befehle, Schiebe- und Rotationsbefehle, ..... ).



## ANMERKUNG

Muß bei Vergleichen zwischen *signed* und *unsigned* unterschieden werden ?

**JA !!**

Beispiel: Byte-Vergleich

unsigned Interpretation:  $11111111_B > 01111111_B$  ( $255_D > 127_D$ )

signed Interpretation:  $11111111_B < 01111111_B$  ( $-1_D < 127_D$ )



### 8.3.5 Bedingte Sprungbefehle für unsigned-Vergleiche mit *cmp*

**cmp Rn, N**      # geht dem Sprungbefehl voraus

Sprungbefehl Bcc	Sprung, wenn gilt	oder anders gesagt
-----	-----	-----
bcs (carry set)	$Rn - N \geq 0$	$Rn \geq N$
bhs (higher or same)	“	“
bhi (higher)	$Rn - N > 0$	$Rn > N$
beq (equal)	$Rn - N = 0$	$Rn = N$
bne (not equal)	$Rn - N \neq 0$	$Rn \neq N$
bcc (carry clear)	$Rn - N < 0$	$Rn < N$
blo (lower)	“	“
bls (lower or same)	$Rn - N \leq 0$	$Rn \leq N$

**Merkregel:** (higher, lower, carry) -Befehle sind unsigned-Vergleiche !



### 8.3.6 Bedingte Sprungbefehle für signed-Vergleiche mit cmp

**cmp rn, N** # geht dem Sprungbefehl voraus

Sprungbefehl Bcc	Sprung wenn gilt	oder anders gesagt
blt (less than)	$Rn - N < 0$	$Rn < N$
ble (less or equal)	$Rn - N \leq 0$	$Rn \leq N$
beq	$Rn - N = 0$	$Rn = N$
bne	$Rn - N \neq 0$	$Rn \neq N$
bge (greater or equal)	$Rn - N \geq 0$	$Rn \geq N$
bgt (greater than)	$Rn - N > 0$	$Rn > N$

**Merkregel:** (greater, less) -Befehle sind signed-Vergleiche !



## Anmerkung zur Implementierung von Verzweigungen und Schleifen

Die im folgenden angegebenen Konstrukte lassen sich mitunter effizienter

- mit bedingten Befehlen oder durch
- Spaghetti-Programmierung (goto)

realisieren.

In dieser Vorlesung (und im Praktikum) wählen wir die wesentlich besser programmierbare und lesbare *strukturierte Assembler-Programmierung*.

Weiterer Vorteil: allgemeingültiger (nicht zu ARM-spezifisch)





### 8.3.7 Implementierung einfacher Verzweigungen

```

if (Bedingung) then
    Anweisungsfolge 1
else
    Anweisungsfolge 2
endif
  
```

Mit Sprungmarken (= Label), wie **if\_nn**, **then\_nn**, **else\_nn** ..... lassen sich hochsprachenähnliche Programmstrukturen (per Konvention) einführen.

Vorteil: erheblich besser lesbar

Anm.: **nn** ist ein frei wählbarer Bezeichner

#### Implementierungsschema 1 (direkte Bed-Auswertung)

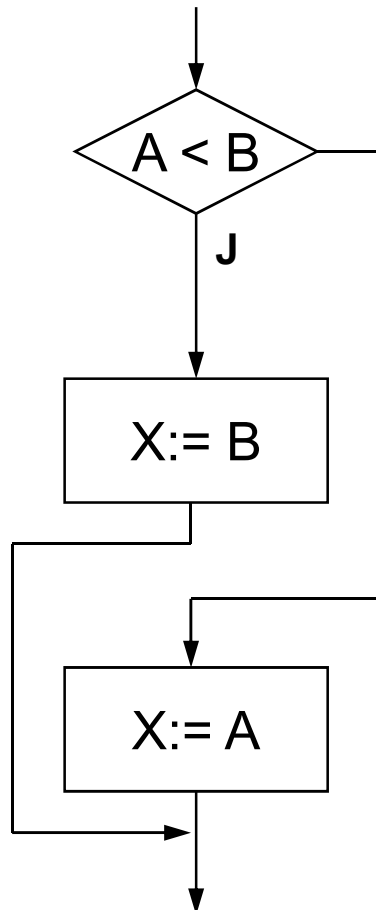
<b>if-nn</b>	<i>Auswertung der Bedg.</i>
	<b>B<sub>bed</sub>    then-nn</b>
	<b>B        else-nn</b>
<b>then-nn</b>	<i>Anweisungsfolge 1</i>
	<b>B        endif-nn</b>
<b>else-nn</b>	<i>Anweisungsfolge 2</i>
<b>endif-nn</b>	

#### Implementierungsschema 2 (negierte Bed-Auswertung)

<b>if-nn</b>	<i>Auswertung der Bedg.</i>
	<b>B<sub>not bed</sub>    else-nn</b>
<b>then-nn</b>	<i>Anweisungsfolge 1</i>
	<b>B        endif-nn</b>
<b>else-nn</b>	<i>Anweisungsfolge 2</i>
<b>endif-nn</b>	



## BEISPIEL: einfache Verzweigung



```

if (A < B) then
    X := B
else
    X := A
end-if
  
```

```

X      DCD      0      ; Ergebnis

      mov      r0, #13   ; A
      mov      r1, #55   ; B
      ldr      r2, =X     ; Zieladresse

; ----- if (A < B) -----
if_01  cmp      r0, r1    ; (A - B) >= 0 ?
      bge      else_01    ; Sprung, wenn A >= B
; ----- A < B -----
then_01 str      r1, [r2]  ; X := B
      b          endif_01
; ----- A >= B -----
else_01 str      r0, [r2]  ; X := A
endif_01
  
```



### 8.3.8 Implementierung von Schleifen

#### *kopfgesteuerte Schleife*

```
while (Laufbedingung)  do
    Anweisungsfolge ....
endwhile
```

#### *fussgesteuerte Schleife*

```
repeat
    Anweisungsfolge .....
until (Abbruchbedingung)
```

#### *Implementierung*

```
while-nn      Auswertung der Bedg.
                Blaufbed  do-nn
                B      endwhile-nn

do-nn
                Anweisungsfolge
                B      while-nn

endwhile-nn
```

#### *Implementierung*

```
repeat-nn      Anweisungsfolge

until-nn      Auswertung der Bedg.
                Babrbed  endrep-nn
                B      repeat-nn

endrepeat-nn
```



### 8.3.9 Implementierung von Zählschleifen

```
for Ifv:= Startwert step Schrittwert until Endwert do  
    Anweisungsfolge ....  
enddo
```

#### *Implementierung*

**for-nn**            *Laufvariable auf Startwert setzen*

**until-nn**        *Laufvariable auf Endwert testen*

**B**<sub>Ifv >= Endwert</sub>      **enddo-nn**

**do-nn**

*Anweisungsfolge*

**step-nn**        *Laufvariable um Schrittwert erhöhen*

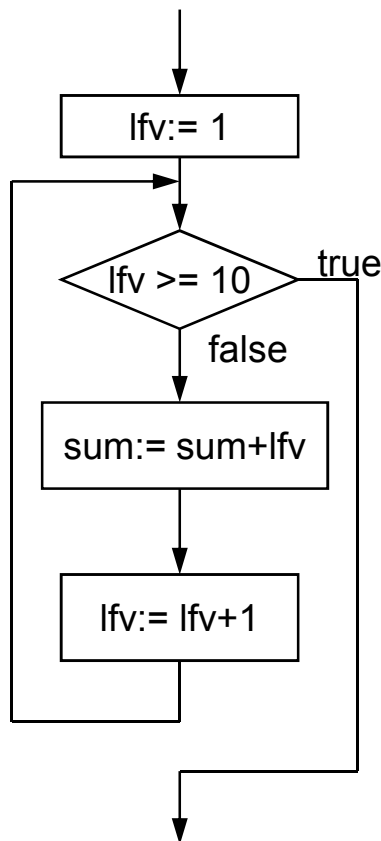
**B**            **until-nn**

**enddo-nn**



## BEISPIEL: Implementierung von Zählschleifen

```
for lfv:= 1  step 1  until 10  do
    sum:= sum + lfv
enddo
```



```

mov  r2, #0           ; sum ← 0
for_01 ; --- INITIALISIERUNG DER SCHLEIFE -----
mov  r0, #1           ; [r0] ← lfv
mov  r1, #10          ; [r1] ← Endw

until_01 ; --- PRÜFUNG DER ABRUCHBEDINGUNG ---
cmp  r0, r1           ; lfv >= Endw.?
bge  enddo_01         ; Sprung wenn lfv >= Endw

do_01 ; --- ANWEISUNGSFOLGE -----
add  r2, r0           ; sum=sum + lfv

step_01 ; --- INCREMENT DER LAUFVARIABLEN -----
add  r0, #1           ; lfv = lfv + 1
b    until_01

enddo_01
```



## ÜBUNG: Zeichen zählen

Gegeben sei ein String, der mit einer 0 abgeschlossen ist.

Es ist ein Assemblerprogramm zu schreiben, welches die Anzahl der „a“ in diesem String zählt.

Gehen Sie wie folgt vor:

1. Pseudocode des Programms erstellen.
2. Assemblerprogramm erstellen.

Register r0 soll als Zeichenzähler verwendet werden.



## ÜBUNG: Stringmanipulation

Es soll ein Assemblerprogramm geschrieben werden, welches aus einem Quellstring einen gleichlangen Zielstring erzeugt, bei dem

1. alle Zahlen sollen durch eine 1 ersetzt und
2. alle anderen Zeichen durch eine 0 ersetzt werden.

Beispiel: Aus dem Quellstring „aBB12xAuo99“  
soll der Zielstring „00011000011“ werden.



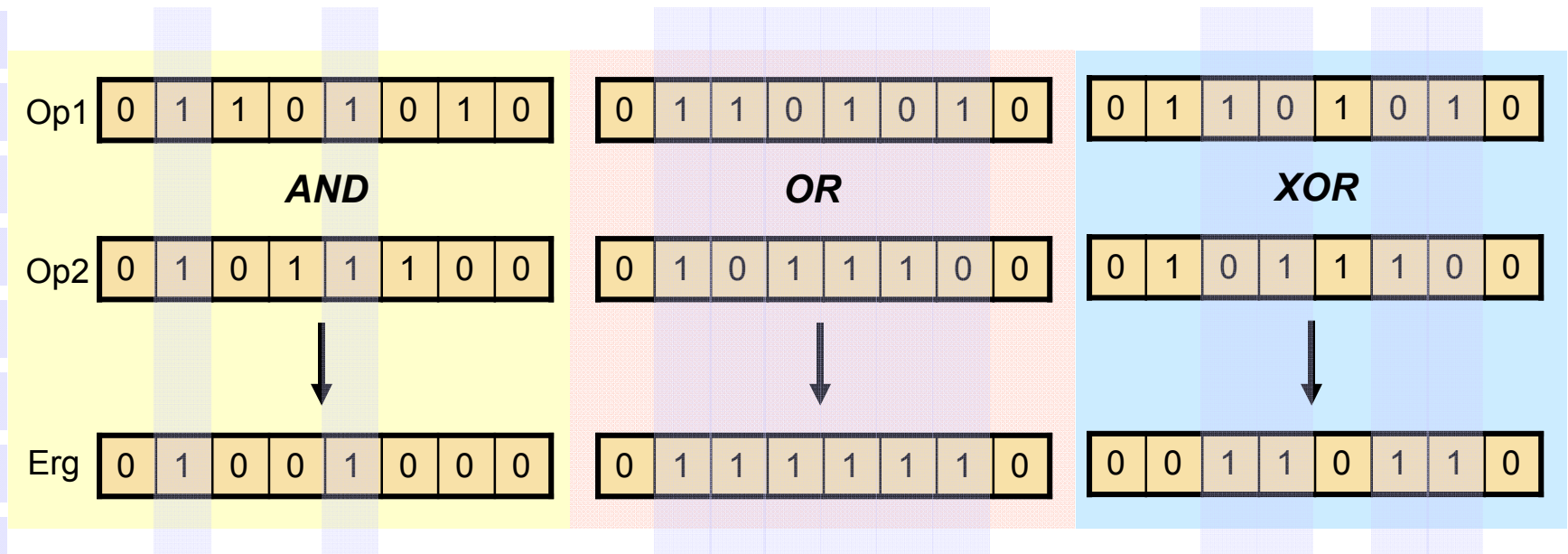
## 8.4 Bitmanipulation

### 8.4.1 Logische Bitoperationen AND, OR, XOR

**AND** = Ergebnis ist 1, wenn beide Operanden 1 sind.

**OR** = Ergebnis ist 1, sobald mindestens einer der beiden Operanden 1 ist.

**XOR** = Ergebnis ist 1, wenn genau einer der beiden Operanden 1 ist.







## 8.4.2 Anwendungsbeispiele

### Programmierung von Ein-/Ausgabe-Karten/Bausteinen/Geräten

- Relaiskarten
- Digital-I/O-Karten
- Schrittmotorkarten
- ADC-Karten (Analog-Digitalwandler)
- Interruptcontroller

### Signalverarbeitung/Bildverarbeitung

- Invertieren
- Graustufenreduktion
- Überlagerung von Bild und Grafik

### Zufallszahlenerzeugung

Datensicherung (CRC-Check)

Verschlüsselung



### 8.4.3 Befehle zur Bitmanipulation

**Aufruf:** <instruction>{<cond>}{S} Rd, Rn, N

<b>AND</b>	Log. bitweises AND zweier 32-bit Werte	$Rd = Rn \& N$
<b>ORR</b>	Log. bitweises OR zweier 32-bit Werte	$Rd = Rn \mid N$
<b>EOR</b>	Log. bitweises XOR zweier 32-bit Werte	$Rd = Rn \wedge N$
<b>BIC</b>	Log. Bit löschen (AND NOT) zweier 32-bit Werte	$Rd = Rn \& \sim N$

#### Beispiele:

- **ands** r0, r1                       $[r0] \leftarrow [r0] \& [r1]$ , Flags passend setzen
- **orr** r0, r1, #0xf0                 $[r0] \leftarrow [r1] \mid 0b111110000$
- **eorne** r0, r0, r1, LSL #8         $[r0] \leftarrow [r0] \wedge [r1] \ll 8$ , falls Z=0



## 8.4.4 Typische Problemstellungen und ihre Lösung

### 8.3.4.1 Extraktion von Bitfeldern

Logische Operationen werden z.B. verwendet, um Bitfelder zu extrahieren

**Beispiel:** Übertragen der **relevanten Bits (11-8)** in **r0**  
und der **relevanten Bits (6-0)** in **r1**  
nach **r2**

<b>r0</b>	1	0	1	1	1	0	1	0	1	0	0	1	1	1	0	0
<b>r1</b>	1	1	0	0	1	0	1	1	0	1	1	0	1	0	0	1
<b>r2</b>	0	0	0	0	1	0	1	0	0	1	1	0	1	0	0	1

Es sei: [r3] = 0xf00 = 2\_0000111100000000

and	r0, r3	; alle Bits löschen, ausser Bits 8-11
and	r1, #0x7F	; alle Bits löschen, ausser Bit (0-6)
orr	r2, r0, r1	; Kombiniert die Bitfelder in r2



### 8.4.4.2 Setzen von Bits

**Beispiel:** Setzen des Bit 10 in r0

Es sei: [r1] = 0x0400 = 2\_0000010000000000

**orr** r0, r1

**r0**

1	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**#0x0400**

0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**r0**

1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



### 8.4.4.3 Löschen von Bits

**Beispiel:** Löschen des Bit 10 in r0

Es sei: [r1] = 2\_ 1111 1111 1111 1111 1111 1011 1111 1111  
[r2] = 2\_ 0000 0000 0000 0000 0000 0100 0000 0000

oder auch:      **and**      r0, r1  
                 **bic**      r0, r2

<b>r0</b>	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0
<b>#0xFBFF</b>	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
<b>r0</b>	1	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0



#### 8.4.4.4 Negieren (toggeln) von Bits

**Beispiel:** Negieren des Bit 10 in r0 (*toggeln* = umschalten)

Es sei: [r1] = 0x0400 = 2\_0000010000000000

**eor r0, r1**

<b>r0</b>	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0
<b>#0x0400</b>	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

<b>r0</b>	1	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

bzw.

<b>r0</b>	1	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0
<b>#0x0400</b>	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

<b>r0</b>	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



## Übung: Bitoperationen (1)

Ein ab Adresse 0x40000000 liegendes 32-Bit-Speicherwort soll wie folgt verändert werden:

Die Bits 0 - 3 sollen auf 1 gesetzt werden,  
die Bits 5 - 9 sollen gelöscht (auf 0 gesetzt) werden und  
die Bits 12 -15 sollen umgeschaltet (getoggelt) werden.

Schreiben Sie ein Assemblerprogramm.



## Übung: Bitoperationen (2)

- a) Schreiben Sie ein Programm "*TestBitPattern*", mit folgendem Verhalten:

[r1] = 1,      wenn Bit 0, 3 und 7 von Register r0 auf 1 gesetzt sind und  
                 alle anderen Bits auf 0,  
         = 0      sonst.

- b) Schreiben Sie ein Unterprogramm "*PatternTester*", mit folgendem Verhalten:

[r0] = 1,      wenn [r1] und [r2] an denjenigen Bitstellen übereinstimmt  
                 die in [r3] mit 1 markiert sind.  
         = 0      sonst.





## 8.5 Rotations- und Verschiebeoperationen

### 8.5.1 Übersicht

Rotations- und Verschiebeoperationen manipulieren den Inhalt einer Speichereinheit durch Verschieben des Bitmusters um einige Stellen nach links oder rechts.

Dazu stehen bei ARM-Prozessor folgende Instruktionen zur Verfügung.

- |                            |            |                     |
|----------------------------|------------|---------------------|
| – Logical Shift left       | <b>lsl</b> | immediate, register |
| – Logical Shift right      | <b>lsr</b> | immediate, register |
| – Arithmetik Shift right   | <b>asr</b> | immediate, register |
| – Rotate right             | <b>ror</b> | immediate, register |
| – Rotate right with extend | <b>rrx</b> | register            |

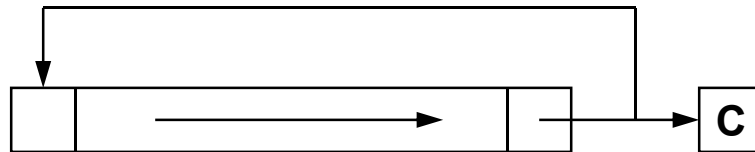


### 8.5.2 Rotation (eines 32-Bit-Wertes nach rechts)

Von einer Rotation wird gesprochen, wenn beim Verschieben kein Bit verloren geht. Die maximale Schiebewert beträgt 31.

Wenn das s-Flag gesetzt ist, wird das herausgeschobene Bit nach C geschrieben.

```
mov    rd, rs, ROR #n
mov    rd, rs, ROR rs2
```



**ror**

#### Beispiele:

```
mov    r0,r1,ROR #3      ; [r1] um 3 Bit nach rechts rotieren
                        ; und in r0 abspeichern

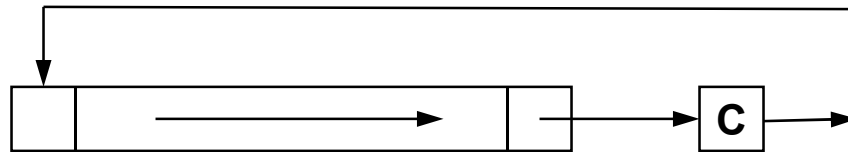
movs   r0,r1,ROR r2      ; [r1] um [r2] Bit nach rechts rotieren
                        ; und in r0 abspeichern, Flags updaten
```



### 8.5.3 Rotation (eines 32-Bit-Wertes nach rechts) um 1 bit über das Carry-Flag

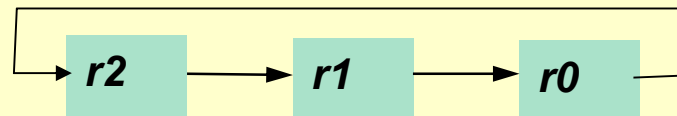
Um eine Rotation über mehr als 32 Bit zu ermöglichen, steht die Instruktion "rrx" zur Verfügung (*rotate through carry*).

```
mov    rd, rs, RRX
```



**rrx**

**Beispiel:** Rotation eines in r0, r1 und r2 stehenden 96-Bit-Feldes um 1 bit



```
movs r3,r0,LSR #1    ; letzte Stelle von r0 → Carry
movs r2,r2,RRX        ; um eine Stelle nach rechts, carry übernehmen
movs r1,r1,RRX        ; um eine Stelle nach rechts, carry übernehmen
movs r0,r0,RRX        ; um eine Stelle nach rechts, carry übernehmen
```



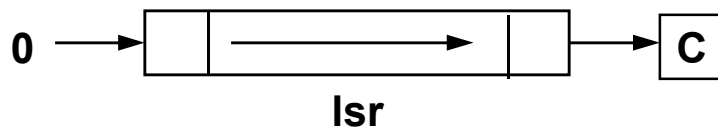
### 8.5.4 Logisches Schieben (eines 32-Bit-Wertes nach links/rechts)

Ein logischer Shift ist eine Verschiebeoperation, wobei in die freiwerdenden Bitpositionen eine 0 nachgeschoben wird.

Wenn das s-Flag gesetzt ist, wird das herausgeschobene Bit nach C geschrieben.

```
mov    rd, rs, LSR #n
mov    rd, rs, LSR rs2
```

```
mov    rd, rs, LSL #n
mov    rd, rs, LSL rs2
```



#### Beispiele:

```
mov    r0, r1, LSR #3      ; [r1] um 3 Bit nach rechts schieben
                                ; und in r0 abspeichern
movs   r0, r1, LSL #2      ; [r1] um 2 Bit nach links schieben
                                ; und in r0 abspeichern, Flags updaten
mov    r0, r1, LSR r2      ; [r1] um [r2] Bit nach rechts schieben
                                ; und in r0 abspeichern
```



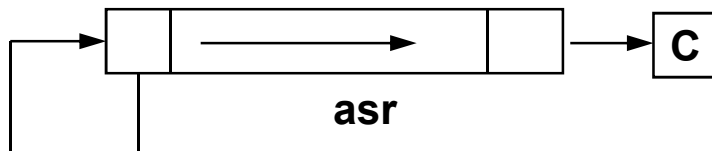
### 8.5.5 Arithmetisches Schieben (eines 32-Bit-Wertes nach rechts)

Ein arithmetischer Shift ist eine Verschiebeoperation auf einem Bitmuster in der 2-er-Komplement-Darstellung (vorzeichenrichtige Erweiterung).

Eine Verschiebeoperation nach links um 1 Bit entspricht der Multiplikation mit 2 und eine Verschiebeoperation nach rechts um 1 Bit entspricht der Division mit 2.

Wenn das s-Flag gesetzt ist, wird das herausgeschobene Bit nach C geschrieben.

```
mov    rd, rs, ASR #n
mov    rd, rs, ASR rs2
```



#### Beispiele:

```
mov    r0,r1,ASR #3      ; [r1] durch 8 dividieren
                        ; und in r0 abspeichern

movs   r0,r1,ASR r2      ; [r1] durch 2^[r2] dividieren
                        ; und in r0 abspeichern, Flags updaten
```



## Übung: Schieben und Rotieren

a) Schreiben Sie in Unterprogramm „*BitCompare*“, mit folgendem Verhalten:

[r2] = 1, wenn Bit 0 und Bit 7 von Register r1 ungleich sind,  
= 0, wenn Bit 0 und Bit 7 von Register r1 gleich sind.

b) Schreiben Sie in Unterprogramm "*RotateLower5*", mit folgendem Verhalten:

- Bei jedem Aufruf des Unterprogramms sollen die Bits 0..4 von *r0* um eine Bitstelle linksherum rotieren.
- Die vorderen Bits 5..31 sollen dabei unverändert bleiben.

