DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

# RISC-Prozessoren am Beispiel ARM: Befehle

# ARM 7 (ARMv3)

- Architektur: ARM v3
- Modelle 1991 – 1993 ARM700, ARM710, ARM710a, ARM7100, ARM7500, ARM7500FE
- 12–40 MHz, 0,889 DMIPS/MHz
- RISC, Load-Store-Architektur
- modifizierte Harvard-Architektur (getrennte Busse zum L1 Daten- und Adress-Cache, aber gemeinsamer weiterer Speicher)
- 3-stufige Pipeline, 16 UserMode-Register, Barrel-Shifter
- Registerbreite: 32 Bit

- *http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt2.pdf*

# ARM Cortex A9 (ARMv7)

- Architektur: ARM v7
- Modelle 2004-2011 ARM Cortex-A (A8, **A9**, A5, A15, A7, A12, A15)
  ARM Cortex-M (M3, M4, M7)
  ARM Cortex-R (R4, R5, R7, R8)

- RISC, Load-Store-Architektur
- modifizierte Harvard-Architektur (getrennte Busse zum L1 Daten- und Adress-Cache, aber gemeinsamer weiterer Speicher)
- NEON Multimedia Befehlssatzerweiterung, SIMD-Befehle
- 8-stufige Pipeline, 16 UserMode-Register, Barrel-Shifter
- Registerbreite: 32 Bit
- enthalten auf vielen SOCs

- *http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt2.pdf*

# ARM Befehlssatz: Generelles

Hauptfeatures des ARM-/ Thumb-Befehlssatzes:

- Die meisten Befehle brauchen nur einen Takt
    - Load/ Store – Architektur
        - Datenverarbeitungsbefehle arbeiten nur mit Registern
            - ALU mit Shifter kombiniert für schnelle Bitmanipulation
        - Spezifische Speicher<->Register -Befehle mit mächtigem Index-Adressierungsmodes
            - flexible Mehrfachregister Load- & Store-Befehle
- Befehlssatzerweiterung SVE2 durch Koprozessoren (für Graphik- (GPUs (Mali)) und KI-Datenverarbeitung (NPU))

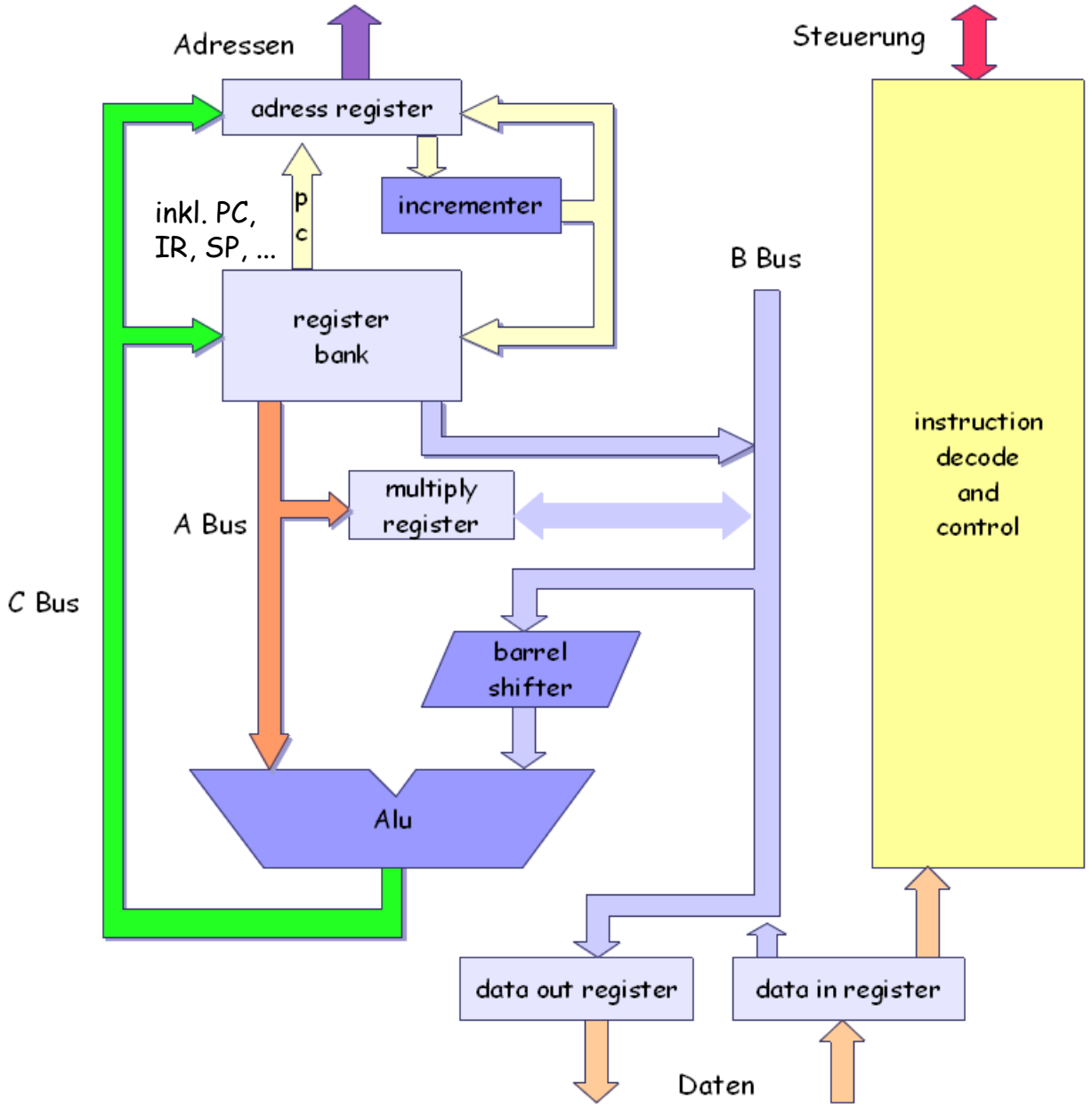
# Befehlssätze/ Execution states: ARM vs. Thumb

| Kriterium | ARM | Thumb |
|---|---|---|
| **Befehlsbreite** | 32 Bit | 16 Bit |
| **Bedingte Befehlsausführung** | Fast immer bedingt | Fast immer unbedingt |
| **Befehlsumfang** | Voller Befehlssatz | Subset des ARM-Befehlssatzes |
| **UAL-Unterstützung** | Unterstützt vom Assembler | Unterstützt vom Assembler |
| **Operandenzahl** | 3 Operanden | 2 Operanden |
| **Speichermodell** | ARMv3 und danach: | Standard: Little Indian, änderbar |

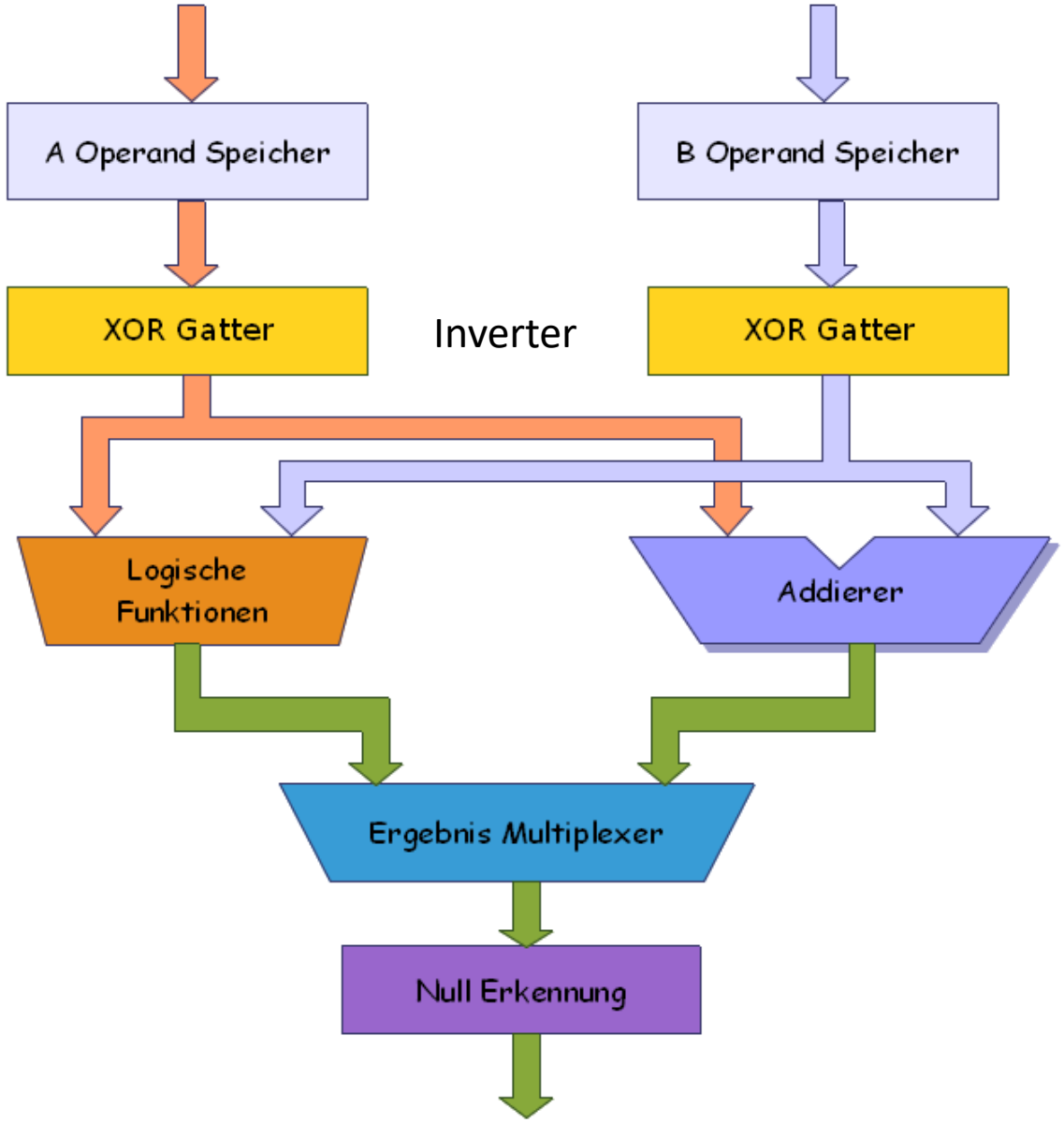
Weitere ARM-Befehlssätze:

- Thumb-2 (16/32 mixed), Jazelle/ DBX for Java bytecode (8 Bit), NEON 64/128-SIMD, VFP vector floating point IS

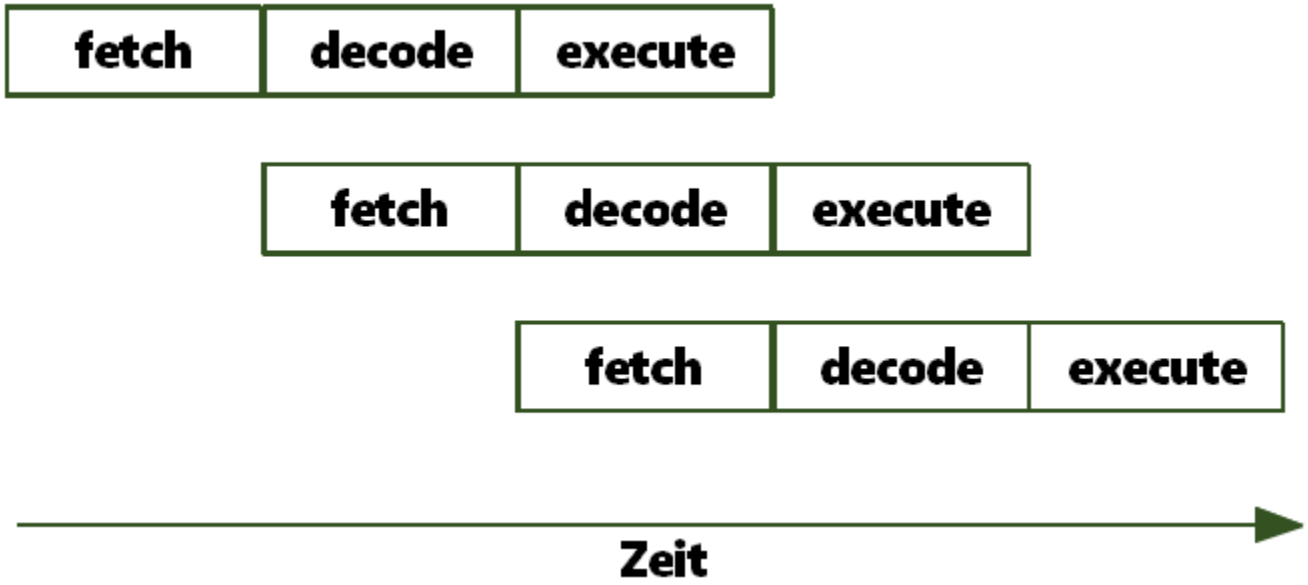
# Die ARM-Datenpfad-Architektur



# Die ARM - ALU


A Operand Speicher
B Operand Speicher
XOR Gatter
Inverter
XOR Gatter
Logische Funktionen
Addierer
Ergebnis Multiplexer
Null Erkennung

# Die ARM7 Pipeline für ALU-Befehle (Operanden: Register)

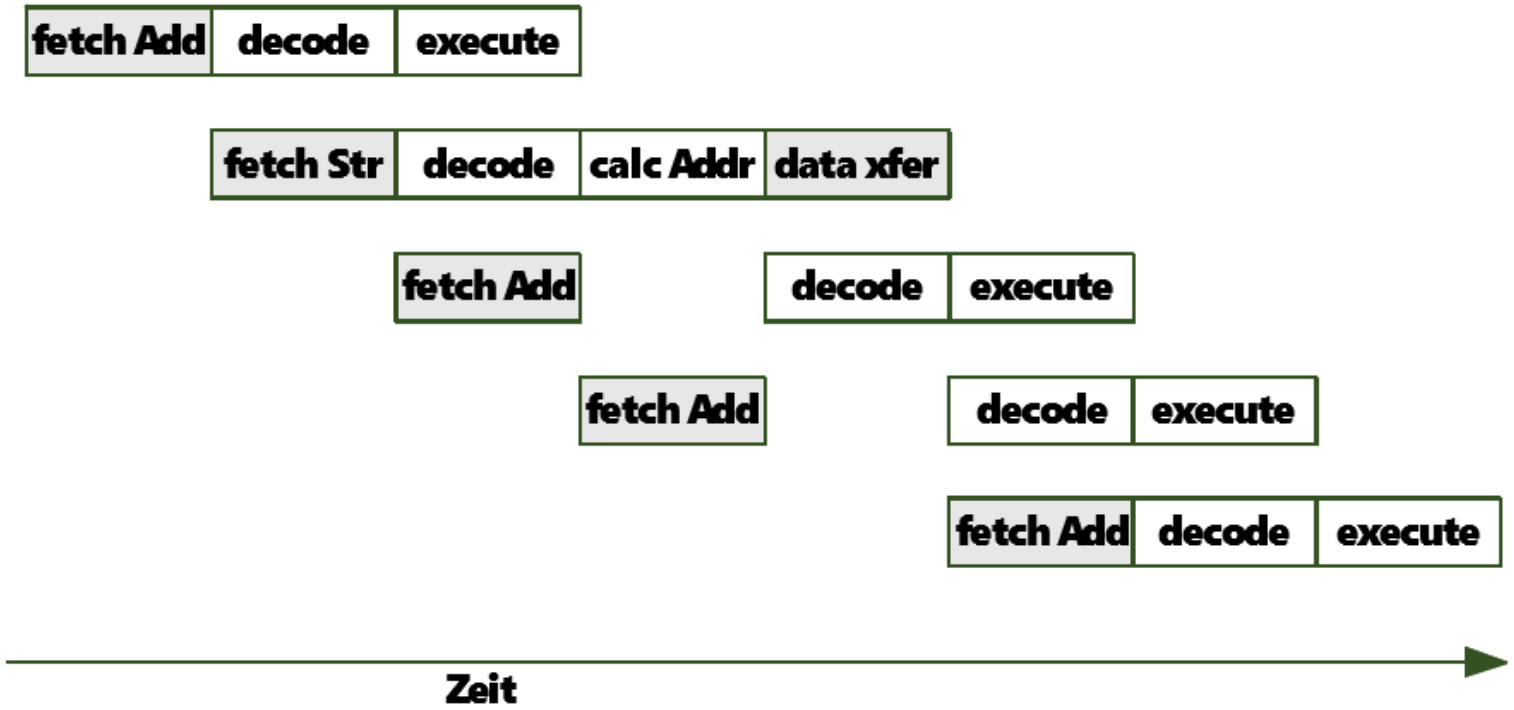



3-stufige Pipeline

Latenz: 3 Zyklen

Durchsatz: 1 Befehl je Zyklus

# Die ARM7-Pipeline für Load/Store-Befehle (Register<->Speicher) (Hazard in von-Neumann-Architekturen)



# ARM Registerstruktur

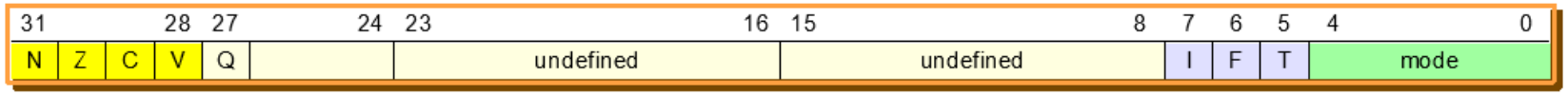
**Anwender-programme** — **Relevant für Exception Handling oder bei der Programmierung auf Systemebene**

| User/System | Fast Interrupt | Interrupt | Supervisor | Undefined | Abort | |
|---|---|---|---|---|---|---|
| R0 | | | | | | Argument/ result/ scratch |
| R1 | | | | | | Argument/ result/ scratch |
| R2 | | | | | | Argument/ scratch |
| R3 | | | | | | Argument/ scratch |
| R4 | | | | | | |
| R5 | | | | | | |
| R6 | | | | | | |
| R7 | R8_fiq | | | | | |
| R8 | R9_fiq | | | | | |
| R9 | R10_fiq | | | | | |
| R10 | R11_fiq | | | | | |
| R11 FP | R12_fiq | | | | | Stack-Frame-Pointer |
| R12 IP | R13_fiq | R13_irq | R13_svc | R13_undef | R13_abt | Intra-ProcureCall-Register |
| R13 SP | R14_fiq | R14_irq | R14_svc | R14_undef | R14_abt | |
| R14 LR | | | | | | |
| R15 PC | | | | | | |
| cpsr | spsr_fiq | spsr_irq | spsr_svc | spsr_undef | spsr_abt | „Current“ ProgramStatusRegister<br>„Saved“ ProgramStatusRegister |

# Big bang theory – fun with flags

https://youtu.be/Xl12Sp1KiEk

# Current Program Status Register (CPSR)




**Condition Code Flags**

- N = **N**egatives ALU Ergebnis
- Z = Alu Ergebnis ist Null (**Z**ero)
- C = Alu Ergebnis erzeugte **C**arry
- V = Alu erzeugte O**v**erflow
- Q = Saturation (Sättigung, 1-stabil)

**Interrupt Disable Bits**

- I = 1, disables IRQ
- F = 1, disables FIQ
  („fast“ Interrupts, maskierbar,
  aber höher prior)

**T-Bit**

- T = 0, Prozessor in Arm State
- T = 1, Prozessor in Thumb State

**Mode Bits**

- Spezifizieren den Prozessor Mode
  (user, system, irq,…)

Termin 3, Aufgabe1.s: ADD->ADDS, Breakpoint auf „MOV R0, R1,…“; R1 im Registersatz CPSR auf 0 setzen
CPSR-Flags ansehen, ADDS-Befehl ausführen, Flags erneut ansehen

# Statusflags bei arithmetischen und logischen Befehlen

- Die Flags des Statusregisters werden **nur** bei den Test- und Compare-Befehlen automatisch gesetzt.
- Alle anderen Befehle müssen die **Statusbits explizit setzen**.
  Dazu ist ein -**S** an den Befehl anzuhängen (Pseudobefehl).
- Beispiel:

  *SUB**S*** *r0, r0, #1*

  *ADD**S*** *r0, r0, #1*

# Bedingte Ausführung von Befehlen

- Jeder Befehl kann bedingt ausgeführt werden. Dazu wird die Bedingung (vorher gesetzte Werte des Statusregisters) als Ergänzung an den Befehl angehängt.
  - Beispiel:

```
        CMP     r0, #5          @ if (r0 != 5)  -  setzt Statusregister

oder:   LDR     r1, #5
        SUBS    r1, r0, r1      @ if (r0-r1 != 0)  -  befehlS setzt Statusregister

        ADDNE r1, r1, r0        @ {    r1 = r1 + r0 -r2
        SUBNE  r1, r1, r2       @ }
                                @ NE = Ergebnis not equal zero
```

- Dies ist eine Besonderheit des ARM32, die viele Tests und Verzweigungen (Sprünge im Programmfluss = neuer Fetch-Zyklus) erspart

# Bedingte Ausführung von Befehlen

| Befehls-ergänzung | Condition Codes Im CPSR | (Letztes Ergebnis) Bedeutung | engl | Codie-rung |
|---|---|---|---|---|
| EQ | Z gesetzt | gleich | equal | 0000 |
| NE | Z gelöscht | ungleich | not equal | 0001 |
| CS/HS | C gesetzt | unsigned >= | carry set / higher or same | 0010 |
| CC/LO | C gelöscht | unsigned < | carry clear / lower | 0011 |
| MI | N gesetzt | negativ | minus | 0100 |
| PL | N gelöscht | positiv oder Null | plus | 0101 |
| VS | V gesetzt | Überlauf | overflow set | 0110 |
| VC | V gelöscht | kein Überlauf | overflow clear | 0111 |
| HI | C gesetzt und Z gelöscht | unsigned > | higher | 1000 |
| LS | C gelöscht und Z gesetzt | unsigned <= | lower or same | 1001 |
| GE | N gleich V | signed >= | greater or equal | 1010 |
| LT | N ungleich V | signed < | less than | 1011 |
| GT | Z gelöscht und N=V | signed > | greater than | 1100 |
| LE | Z gesetzt oder N!=V | signed <= | less or equal | 1101 |
| AL | Always | Default | always | 1110 |

# Grundsätzliches zu ARM

Drei Haupt-Befehlstypen

- Befehle zur Datenverarbeitung (z.B. Addition)
- Ablaufsteuerung (z.B. Verzweigungen)
- Befehle zur Adressierung (z.B. Speicherzugriffe)

Regeln:

- Alle Operanden sind 32 Bit (Register, Konstanten,…)
- Das Ergebnis ist 32 Bit und wird in einem Register abgelegt
- 3 (optionale) Operanden: 2 als Input,  1 Operand für das Ergebnis

Beispiel:

```
ADD  R0, R1, R2          ; R0 := R1 + R2
```

# Befehlscodierung

L=1: load (0: store) / Branch with Link
U=1: up, add offset to base (0: down)
P=1: pre, add offset before transfer (0: post)
I(DP)=1: Reg/ use barrel shifter
I(LS)=0: offset is immediate value
S=1: Set condition codes
A=1: accumulates last operand
W=1: write address into base (0: no write-back)
B=1: transfer byte quantity (0: word)

Arithmetische und logische Datenverarbeitung (nur Register)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Instruction Type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Condition | | | | 0 | 0 | I | OPCODE | | | | S | Rn | | | | Rd | | | | OPERAND-2 | | | | | | | | | | | | | Data processing |
| Condition | | | | 0 | 0 | 0 | 0 | 0 | 0 | A | S | Rd | | | | Rn | | | | Rs | | | | 1 | 0 | 0 | 1 | Rm | | | | Multiply |
| Condition | | | | 0 | 0 | 0 | 0 | 1 | U | A | S | Rd HIGH | | | | Rd LOW | | | | Rs | | | | 1 | 0 | 0 | 1 | Rm | | | | Long Multiply |
| Condition | | | | 0 | 0 | 0 | 1 | 0 | B | 0 | 0 | Rn | | | | Rd | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rm | | | | Swap |

## Operation Code

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

# Befehlscodierung

L=1: load (0: store) / Branch with Link
U=1: up, add offset to base (0: down)
P=1: pre, add offset before transfer (0: post)
I(DP)=1: use barrel shifter (Reg)
I(LS)=0: offset is immediate value
S=1: Set condition codes
A=1: accumulates last operand
W=1: write address into base (0: no write-back)
B=1: transfer byte quantity (0: word)

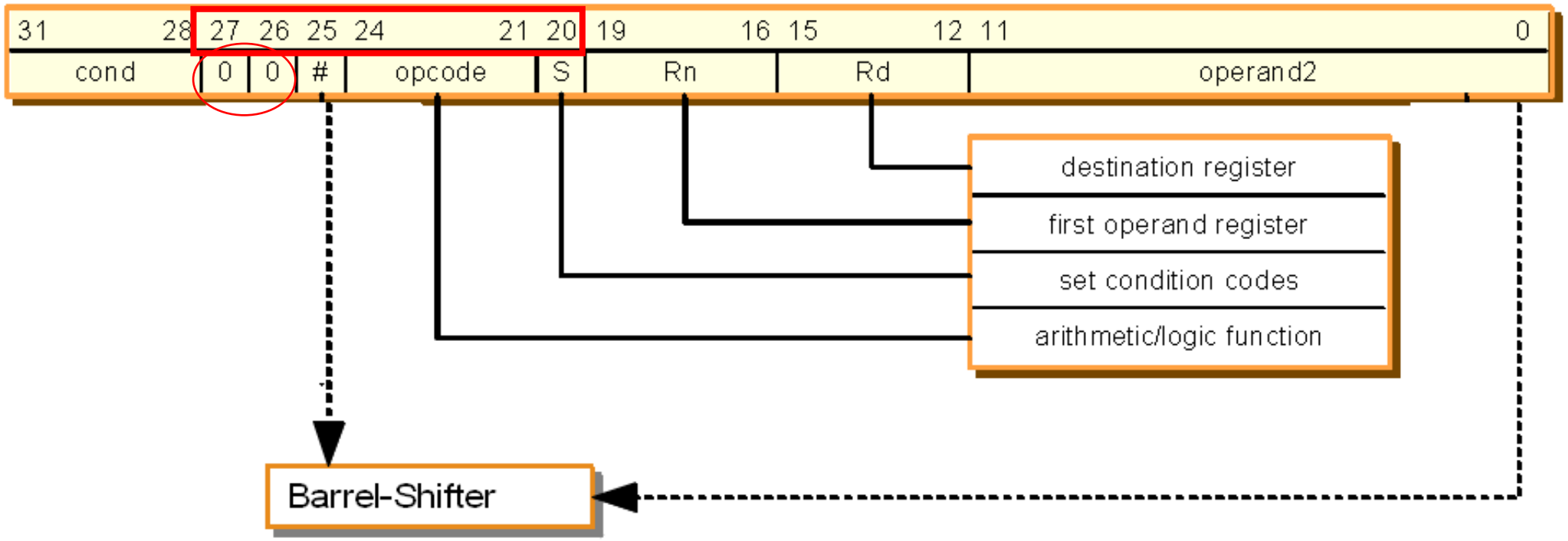
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Instruction Type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arithmetische und logische Datenverarbeitung (nur Register) | Condition | | | | 0 | 0 | I | OPCODE | | | | S | Rn | | | | Rs | | | | OPERAND-2 | | | | | | | | | | | | Data processing |
| | Condition | | | | 0 | 0 | 0 | 0 | 0 | 0 | A | S | Rd | | | | Rn | | | | Rs | | | | 1 | 0 | 0 | 1 | Rm | | | | Multiply |
| | Condition | | | | 0 | 0 | 0 | 0 | 1 | U | A | S | Rd HIGH | | | | Rd LOW | | | | Rs | | | | 1 | 0 | 0 | 1 | Rm | | | | Long Multiply |
| | Condition | | | | 0 | 0 | 0 | 1 | 0 | B | 0 | 0 | Rn | | | | Rd | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rm | | | | Swap |
| Speicher-adressierung (LOAD/ STORE) | Condition | | | | 0 | 0 | 0 | P | U | 1 | W | L | Rn | | | | Rd | | | | OFFSET 1 | | | | 1 | S | H | 1 | OFFSET 2 | | | | Halfword Transfer Imm Off |
| | Condition | | | | 0 | 0 | 0 | P | U | 0 | W | L | Rn | | | | Rd | | | | 0 | 0 | 0 | 0 | 1 | S | H | 1 | Rm | | | | Halfword Transfer Reg Off |
| | Condition | | | | 0 | 1 | I | P | U | B | W | L | Rn | | | | Rd | | | | OFFSET | | | | | | | | | | | | Load/Store - Byte/Word |
| | Condition | | | | 1 | 0 | 0 | P | U | B | W | L | Rn | | | | REGISTER LIST | | | | | | | | | | | | | | | | Load/Store Multiple |
| Ablaufsteuerung, Sprungbefehle | Condition | | | | 1 | 0 | 1 | L | BRANCH OFFSET | | | | | | | | | | | | | | | | | | | | | | | | Branch |
| | Condition | | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rn | | | | Branch Exchange |
| | Condition | | | | 1 | 1 | 0 | P | U | N | W | L | Rn | | | | CRd | | | | CPNum | | | | OFFSET | | | | | | | | COPROCESSOR DATA XFER |
| | Condition | | | | 1 | 1 | 1 | 0 | Op-1 | | | | CRn | | | | CRd | | | | CPNum | | | | OP-2 | | | 0 | CRm | | | | COPROCESSOR DATA OP |
| | Condition | | | | | | | | OP-1 | | | L | CRn | | | | Rd | | | | CPNum | | | | OP-2 | | | 1 | CRm | | | | COPROCESSOR REG XFER |
| | Condition | | | | 1 | 1 | 1 | 1 | SWI NUMBER | | | | | | | | | | | | | | | | | | | | | | | | Software Interrupt |

# ARM-Prozessor

Drei Befehlstypen

- **Befehle zur Datenverarbeitung (z.B. Addition, Multiplikation)**
- Ablaufsteuerung (z.B. Verzweigungen)
- Befehle zur Speicheradressierung (LOAD/ STORE)

# Arithmetische und logische Befehle




Binärcodierung der Datenverarbeitungsbefehle

# Assembler-Format

Arithmethische und logische Befehle im Assembler-Darstellung haben eines der beiden folgenden Formate:

| | | |
|---|---|---|
| *<op>{<cond>}{S}* | *Rd, Rn, #<32-Bit-Immidiate>* | *@ destination, operand1, operand2* |
| *<op>{<cond>}{S}* | *Rd, Rn, Rm {,<shift>}* | *@ destination, operand1, operand2, opt. shift op2* |

Rd: Destination
Rn: Operand 1 (befehlsabhängig)
Rm: Operand 2 (optional)

```
Beispiel: ADD     r0, r1, r2          @ ADD ohne Setzen der Flags
          ADDGTS  r0, r1, r2, LSR r3  @ ADD nur bei GT, mit Setzen der Flags (shift um low-Byte-Wert des r3)
          ADD     r0, r1, #4
          MOV     r0, r1              @ logischer Befehl
```

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

# Arithmetische und logische Befehle

| Typ | Befehl | Operation | Bedeutung | Opcode |
|---|---|---|---|---|
| | AND | Rd =Op1 and Op2 | and | 0000 |
| | EOR | Rd =Op1 xor Op2 | exclusive or | 0001 |
| | ORR | Rd =Op1 or Op2 | inclusive or | 1100 |
| | BIC | Rd =Op1 and not Op2 | bit clear, Maskierung | 1110 |
| | ADD | Rd =Op1 +Op2 | add | 0100 |
| | ADC | Rd =Op1 +Op2 +Carrybit | add with carry | 0101 |
| | SUB | Rd =Op1 - Op2 | substract | 0010 |
| | SBC | Rd =Op1 – Op2 +Carrybit - 1 | sub with carry | 0110 |
| | RSB | Rd =Op2 – Op1 | reverse substract | 0011 |
| | RSC | Rd =Op2 – Op1 +Carrybit - 1 | reverse sub with carry | 0111 |
| | TST | setzt CC bzgl. Op1 AND Op2 | (bit) test | 1000 |
| | TEQ | setzt CC bzgl. Op1 EOR Op2 | test equivalence | 1001 |
| | CMP | setzt CC bzgl. Op1 - Op2 | compare | 1010 |
| | CMN | setzt CC bzgl. Op1 + Op2 | compare negated | 1011 |
| | MOV | Rd =Op2 | move | 1101 |
| | MVN | Rd =not op2 | move bitweise invertiert | 1111 |

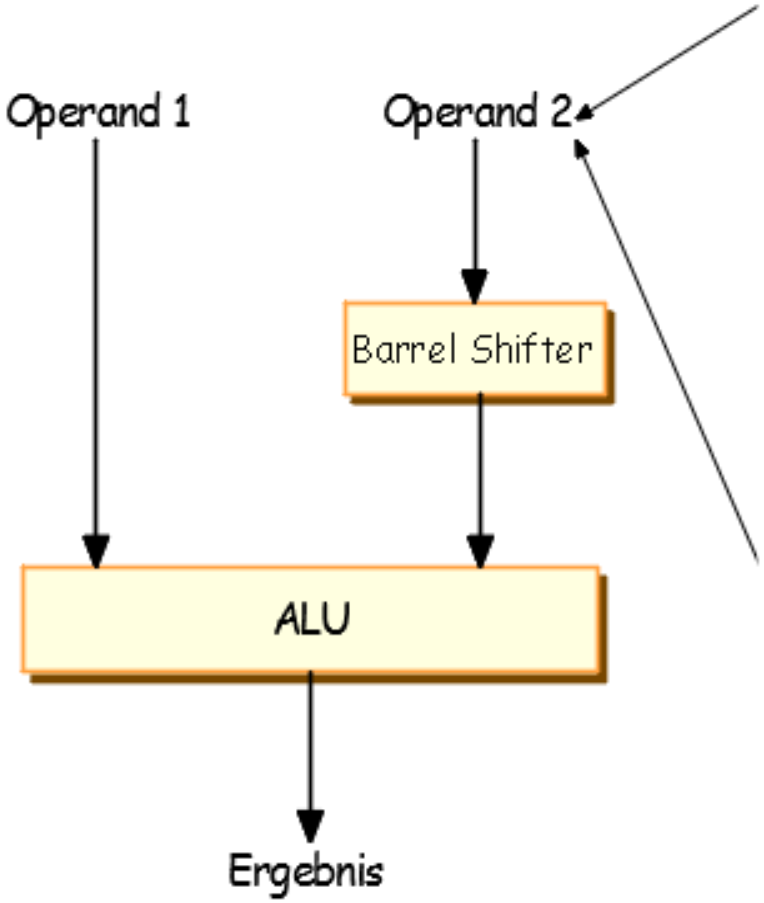
Logische Op. Arithm. Op. Vergleich Schiebeop.

r1= #1 r2=#2 // vor jedem Befehl

```
AND  r0, r1, r2  // r0=0 , keine Flags
EOR  r0, r1, r2  // r0=3 , keine Flags
ORR  r0, r1, r2  // r0=3 , keine Flags
BIC  r0, r1, r2  // r0=1 , keine Flags
ADD  r0, r1, r2  // r0=3 , keine Flags
ADC  r0, r1, r2  // r0=4 bei CY=1
SUB  r0, r1, r2  // r0=-1 , keine Flags
SBC  r0, r1, r2  // r0=-2 bei CY=0
RSB  r0, r1, r2  // r0=1, keine Flags
RSC  r0, r1, r2  // r0=0 bei CY=0
TST  r1, r2      // Flags wie ANDS, NZCV=0100
TEQ  r1, r2      // Flags wie EORS, NZCV=0000
CMP  r1, r2      // Flags wie SUBS, NZCV=1000
CMN  r1, r2      // Flags wie ADDS, NZCV=0000
MOV  r1, r2      // r1=2, keine Flags
MVN  r1, r2      // r1=0xFFFFFFFD
```

CC = condition codes

# Benutzung des Barrel Shifters




**Operand 2 ist ein Register**, zusätzlich kann eine Shift-Operation angewandt werden.

Shift-Werte können sein:

- 5 Bit unsigned integer (0...31 Shifts)
- Wert in einem anderen Register (niedrigstes Byte, extra Takt)

Wird benutzt zur Multiplikation des Op2 mit Konstanten.

```
ADD  r0, r1, r2, LSL r3
SUB   r0, r1, r2, LSL #6
```

**Operand 2 ist ein Immediate Wert** + Rotate Operation

- 8-Bit-Zahl im Bereich zwischen 0 und 255
- Kann nach rechts rotiert werden um gerade Zahl (s. Folie 27)

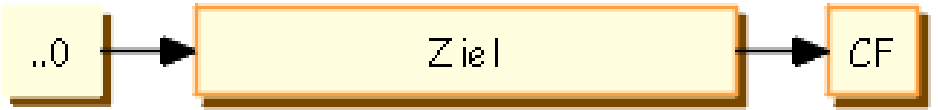
# Der Barrel Shifter (logical shift)
# LSL...: Pseudobefehle für MOV mit Shift

LSL: Logical Shift Left

nur bei „LSLS“!


Multiplikation mit einer Zweier Potenz

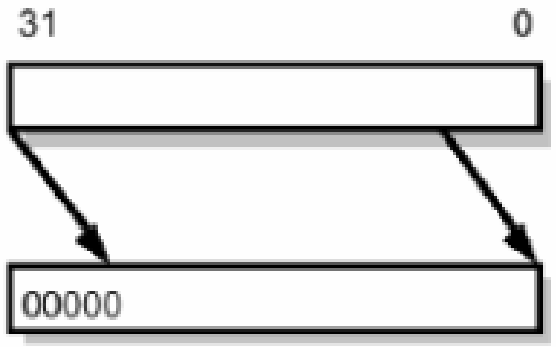
LSR: Logical Shift Right




nur bei „LSRS“!

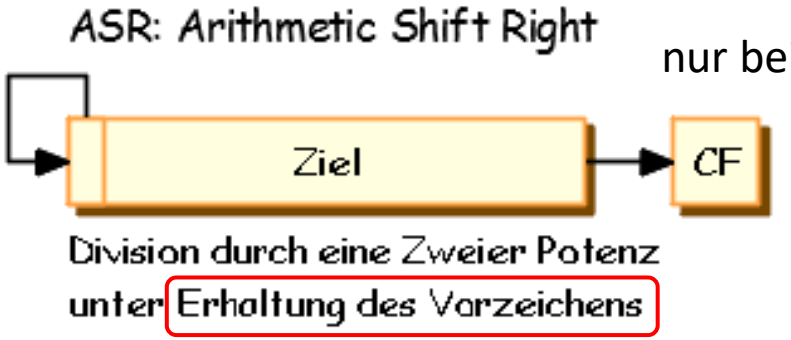
Division durch eine Zweier Potenz


LSL #5



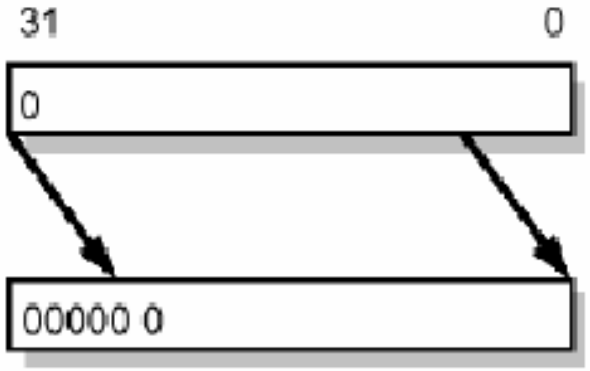

LSR #5

# Der Barrel Shifter (arithmetic shift)

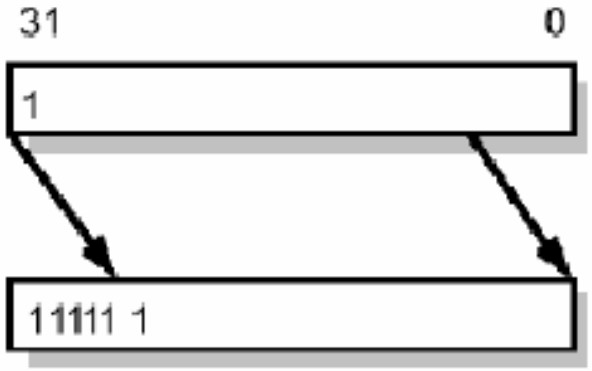



nur bei „ASRS“!

ASL = LSL

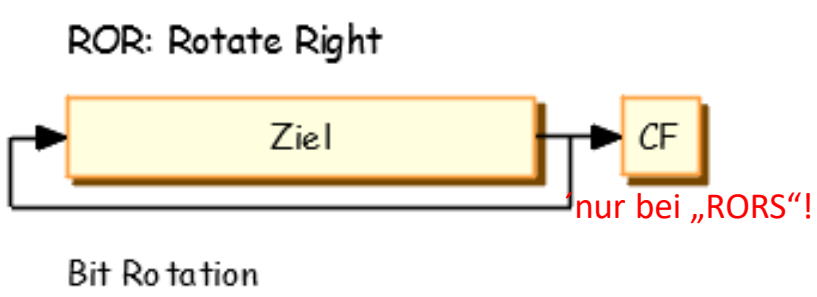


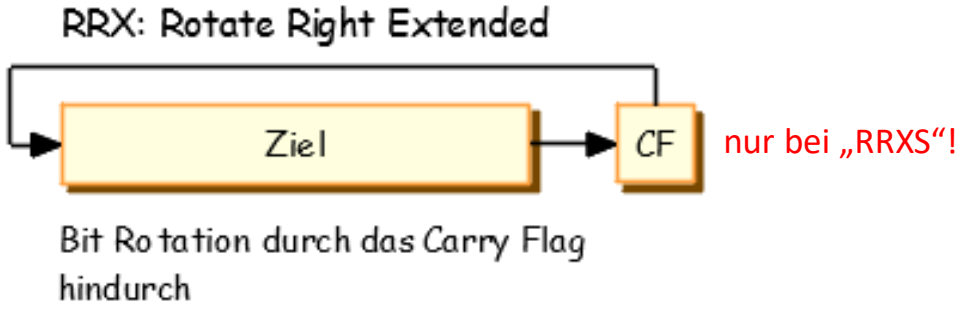

ASR #5, positive operand

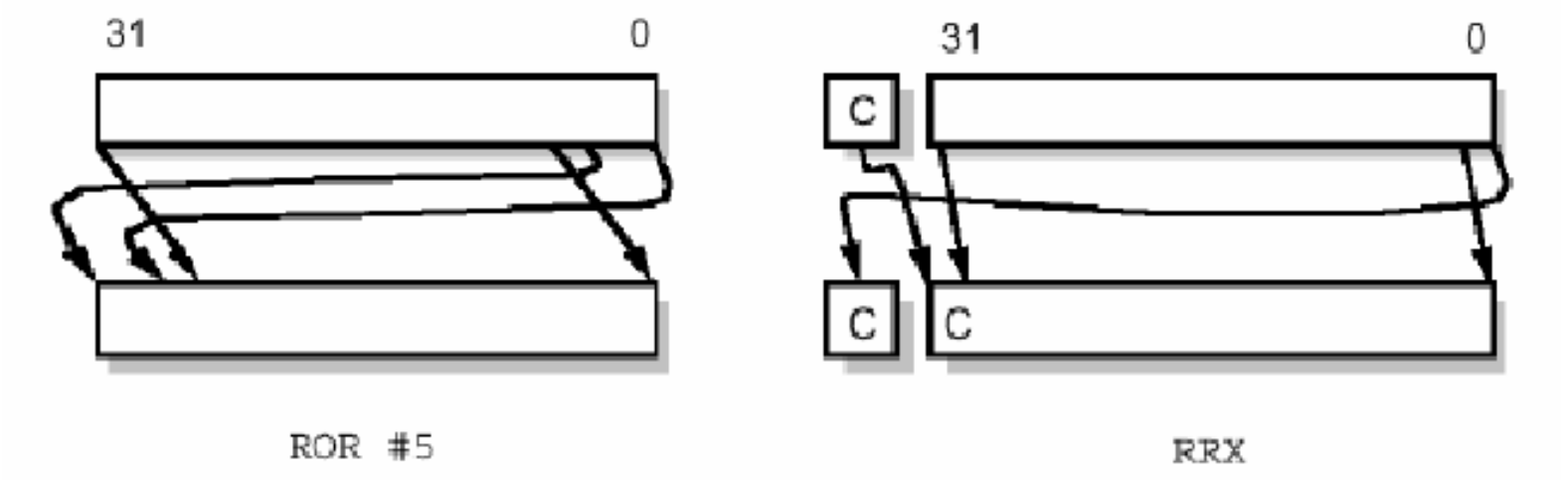



ASR #5, negative operand

# Der Barrel Shifter (rotate)



# Multiplikation mit einer Konstanten

- Multiplikationen mit Konstanten, die sich als $2^x$ oder als $2^x \pm 1$ darstellen lassen, können in einem Zyklus durchgeführt werden
- Beispiel: Multiplikation von r1 mit 5

```
add r0, r1, r1, LSL #2   // r0=r1+r1*4
```

- Durch Zusammensetzen mehrerer Instruktionen können auch komplexere Multiplikationen durchgeführt werden
- Beispiel: Multiplikation von r1 mit 10

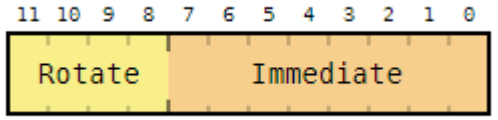
```
add r0, r1, r1, LSL #2          // r0 = r1 + r1*4
mov r0, r0, LSL #1;             // r0 = r0*2
```

- Beipiel Multiplikation von r3 mit 119 = 17 * 7 = (16 + 1) * (8 - 1)

```
add r2, r3, r3, LSL #4          // r2 = r3 + r3 * 16 = r3*17      r2=r3*17
rsb r2, r2, r2, LSL #3          // r2 = r2 * 8 - r2 = r2*7        r2=r2*7
```

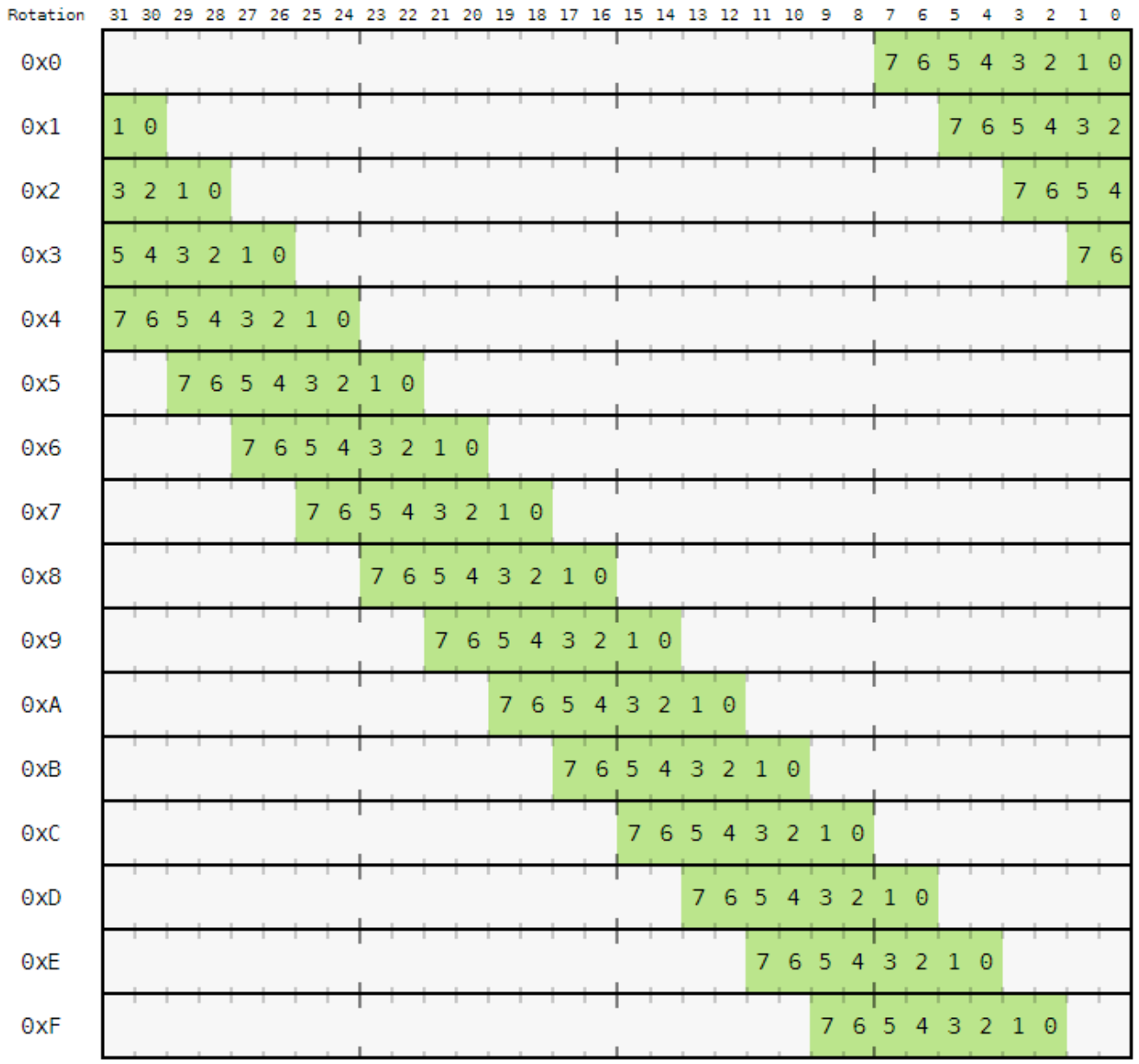
# Operand 2 als Konstante (immediate value)

- Das Befehlsformat für Datenverarbeitungsbefehle reserviert 12 Bits für Operand 2 (8 Bit Konstante, 4 Bit optionale Rechtsrotation)
    - 12 Bit ergäben einen Wertebereich von max. $2^{12} = 4096$
    - Genutzt: Wertebereich mit **8 Bits** ( 0 – 255 ) als 32-Bit-Wert (führende Nullen)
      + Rotation (**4 Bit** * 2 (implizit): um 2,4,6,8,… Stellen, = *4, *16, *64…)
- Diese 8 Bit-Werte (24 führende Nullen) können (rechts) rotiert werden um eine geraden Anzahl von Positionen (ROR mit 0, 2, 4,..30).
- Das ergibt eine weitaus größere Abdeckung des Zahlenbereiches: *8 zusammenhängende Bits innerhalb des 32-Bit-Wertebereichs!*
- Je größer die Werte, desto größer die Lücken (ungenauere Werte)
- Einige Konstanten müssen dennoch vom Speicher geladen werden oder müssen in einem Register konstruiert werden.



# Abbildbare immediate Werte

(je größer die Werte, desto größer die wertmäßigen Lücken)



# Operand 2 als Konstante (immediate value)

- Wertebereiche:

| | | |
|---|---|---|
| 0 - 255 | [0 – 0xff] | @ Distanz: 0 |
| 256,260,264,..,1020 | [0x100-0x3fc, step 4, 0x40-0xff ror 30] | @ Distanz: 4 |
| 1024,1040,1056,..,4080 | [0x400-0xff0, step 16, 0x40-0xff ror 28] | @ Distanz: 16 |
| 4096,4160, 4224,..,16320 | [0x1000-0x3fc0, step 64, 0x40-0xff ror 26] | @ Distanz: 64 |

  ...

- Beispiel MOV – Befehl:

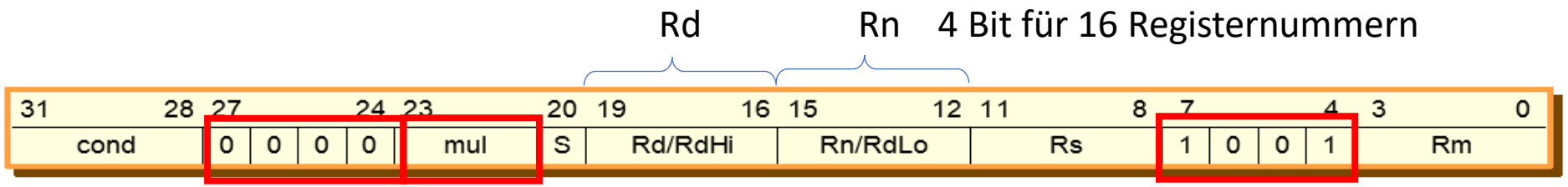
  MOV r0, #0xFF000000 ; umgesetzt im Befehlcode als => MOV r0, (#0xFF ROR $2*2^2$)

- Beispiel MVN: erzeugt das bitweise Komplement (1er Komplement)

  MOV r0, #0xFFFFFFFF ; umgesetzt im Befehl als MVN r0, #0

- Falls die benötigten Konstanten nicht erzeugt werden können, erzeugt der Assembler eine Fehlermeldung! Uff!

# Multiplikation (nur Register-Register!)




Assembler Format:

| | | |
|---|---|---|
| MUL{<cond>}{S} | Rd, Rm, Rs | @ mul=000 |
| MLA{<cond>}{S} | Rd, Rm, Rs, Rn | @ mul=001 |
| <mul>{<cond>}{S} | RdHi, RdLo, Rm, Rs | |

mul:

| Opcode [23:21] | Mnemonic | Bedeutung | Effekt |
|---|---|---|---|
| 000 | MUL | Multiply (32 Bit Ergebnis) | Rd := (Rm * Rs) [31:0] |
| 001 | MLA | Multiply-accumulate (32 Bit Ergebnis) | Rd := (Rm * Rs + Rn) [31:0] |
| 100 | UMULL | Unsigned multiply long | RdHi:RdLo := Rm * Rs |
| 101 | UMLAL | Unsigned multiply-accumulate long | RdHi:RdLo += Rm * Rs |
| 110 | SMULL | Signed multiply long | RdHi:RdLo := Rm * Rs |
| 111 | SMLAL | Signed multiply-accumulate long | RdHi:RdLo += Rm * Rs |

Ergebnis wird auf 2 Register aufgeteilt (64 Bit)

# Übung 1 – bitte im Chat antworten

1. Welche Befehle implementieren folgende Zuweisungen (ohne explizite Multiplikation):
   a) r0 = 16
   b) r1 = r0 * 4
   c) r0 = r1 / 16 ( r1: 2er-Komplement mit Vorzeichen)
   d) r1 = r2 * 7
2. Was machen folgende Befehle ?
   a) ADDS r0, r1, r1, LSL #2 b) RSB r2, r1, #0
3. Was ergibt folgende Befehlssequenz ?

```
ADD   r0, r1, r1, LSL #1
SUB   r0, r0, r1, LSL #4
ADD   r0, r0, r1, LSL #7
```

# Übung 2

r0 = 0xAABBCCDD

- EOR r1, r0, r0, ror #16
- BIC r1, r1, #0xFF0000
- MOV r0, r0, ror #8
- EOR r0, r0, r1, lsr #8

r0 = ?

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

# Fragen zu den DV-Befehlen?

# Grundsätzliches zu ARM

Drei Befehlstypen

- Befehle zur Datenverarbeitung (z.B. Addition)
- **Ablaufsteuerung (z.B. Verzweigungen)**
- **Befehle zur** zur Speicheradressierung (LOAD/ STORE)

# Befehle zur Ablaufsteuerung

**Sprungbefehle (Branch):**

Ausführung wird dauerhaft an einer anderen Adresse fortgesetzt.

**Branch-and-Link-Befehle (entspricht Call):**

eine Rücksprungadresse wird gespeichert, so dass an die ursprüngliche Programmsequenz zurückgesprungen werden kann.

**Supervisor-Calls:**

Wechsel in den geschützten Systemcode, z.B. für Zugriffe auf Register von Peripherie

Link-Register: bekommt bei Calls die Returnadresse , um deren aufwändiges Push und Pop in den Speicher zu vermeiden.
Achtung: bei verschachtelten Aufrufen vor Call (BL) explizit auf den Stack sichern, und nach Rückkehr wieder herstellen!

# Assembler-Format

Assembler-Formate der Sprungbefehle:

| | | |
|---|---|---|
| B{L}{<cond>} | <Zieladresse> | Branch w/o Linkr. (mit oder ohne Füllen des Link registers aus dem PC)<br>B<cond> = (un-)bedingter Sprung (JMP oder JNE oder JGE oder …<br>BL<cond> = (un-)bedingter Call |
| B{L}X{<cond>} | Rm | Branch with optional Exchange w/0 Linkregister<br>= Jump oder Call mit erlaubtem Wechsel des Modes (ARM<->Thumb) |
| BLX | <Zieladresse> | Branch with Exchange and Linkregister<br>= Call mit Modewechsel in Thumb, akt. Mode wird ins LSB des LR gerettet (unterste 2 Bit sind 0, wenn aligned, LSB ist immer 0, da 16-Bit-Adressen immer gerade sind) |

BLX nur verfügbar in den T-Varianten ab ARM v5!

# Befehle zur Ablaufsteuerung

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Instruction Type |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Condition | | | | 1 | 0 | 1 | L | BRANCH OFFSET | | | | | | | | | | | | | | | | | | | | | | | | Branch |
| Condition | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rn | | | | | | | Branch Exchange |

L: 1= Branch with Link, 0= Branch

Binärcodierung der Ablaufsteuerungsbefehle

Codierung für Branch: B, BL Zieladresse

Codierung für Branch mit optionalem Exchange: BX Rn (LSB bestimmt Exchange)

BLX Rn | Zieladresse

# Sprungbefehle (Branch)

- Sprungbefehle dienen dazu, den Kontrollfluss von Programmen zu verändern.
- In der Kombination mit Vergleichsbefehlen können mit Ihnen alle wichtigen Kontrollflussanweisungen wie „if.. else“, „while“, „for“ oder „switch“ realisiert werden.
- Das Ziel einer Sprung-/Callanweisung bei B/ BL ist immer eine Marke (Label), der Assembler erzeugt die Sprungadresse (PC-relativ, in 26 (24+2) Bit, niedrigste beide Bit nicht codiert, da immer =0: +- 32 MB relativ zum PC).
- Beispiel:

```
          CMP R0, #0
          BNE Marke1
                …
                …
Marke1:   MOV R5, #5
                …
```

# Sprungbefehle (auch bedingt)

| Branch | Interpretation | Normal uses |
|---|---|---|
| B | Unconditional | Always take this branch |
| BAL | Always | Always take this branch |
| BEQ | Equal | Comparison equal or zero result |
| BNE | Not equal | Comparison not equal or non-zero result |
| BPL | Plus | Result positive or zero |
| BMI | Minus | Result minus or negative |
| BCC | Carry clear | Arithmetic operation did not give carry-out |
| BLO | Lower | Unsigned comparison gave lower |
| BCS | Carry set | Arithmetic operation gave carry-out |
| BHS | Higher or same | Unsigned comparison gave higher or same |
| BVC | Overflow clear | Signed integer operation; no overflow occurred |
| BVS | Overflow set | Signed integer operation; overflow occurred |
| BGT | Greater than | Signed integer comparison gave greater than |
| BGE | Greater or equal | Signed integer comparison gave greater or equal |
| BLT | Less than | Signed integer comparison gave less than |
| BLE | Less or equal | Signed integer comparison gave less than or equal |
| BHI | Higher | Unsigned comparison gave higher |
| BLS | Lower or same | Unsigned comparison gave lower or same |

# Unterprogramme (Branch-with-Link)

- Der Befehl BL (branch-with-link) führt einen Sprung aus und speichert die Rücksprungadresse im Link-Register r14 (LR) = Call.

- Weitere Subroutinen müssen r14 (LR) vor dem Call im Speicher sichern, z.B. auf dem Stack, und nach erfolgter Rückkehr wieder herstellen.
  Arbeitsregister (>r3) sollten ebenfalls gesichert werden, wenn das Unterprogramm diese benötigt.

- Beispiel:

```
        PUSH  LR          @ wenn man selbst ein UP ist und zurückkehren möchte
        BL  SUBR          @ Sprung zu SUBR
        POP  LR           @ Rücksprung hierher
          …
SUBR:     …               @ Eintrittspunkt der Subroutine
        MOV pc, lr        @ Return
```

(besser: BX LR, da Thumb PC im MOV-Befehl nicht zulässt)

# (A)APCS Konvention zur Parameterübergabe

- ARM (Architecture) Procedure Call Standard
- Die Parameter 1-4 werden beim Unterprogrammaufruf in den Registern r0-r3 übergeben (Zeit und Stackplatz sparen).
- Der Rückgabewert von Funktionen steht im Register r0 oder r0/r1.
- Die Register r0-r3 und das Register ip sind Scratch Register und deren Inhalte dürfen im Unterprogramm zerstört werden.
- Die Informationen in r4-r10, fp, sp müssen erhalten bleiben.
- der Inhalt von lr wird zur Rückkehr ins aufrufende Programm benötigt – nicht zerstören!

| | |
|---|---|
| R0 | ARG1/Result |
| R1 | ARG2/Scratch |
| R2 | ARG3/Scratch |
| R3 | ARG4/Scratch |
| R4 | Var1 |
| R5 | Var2 |
| R6 | Var3 |
| R7 | Var4 |
| R8 | Var5 |
| R9 | Var6 |
| R10 | Var7 |
| FP | FP |
| IP | IP/Scratch |
| SP | SP |
| LR | LR |
| PC | PC |

# Sprungbefehle mit Modewechsel

- BX Rn **Branch and Exchange**

Als Argument erhält der Befehl eine Registeradresse. Der Inhalt der Registeradresse wird in den Programmcounter (r15) kopiert (niederwertigstes Bit ist sowohl bei ARM als auch Thumb-Mode=0, da mindestens 16-Bit aligned – dieses Bit wird benutzt zum Speichern des T-Mode-Bits). Ist die Zieladresse also ungerade, erfolgt ein Wechsel in den Thumbmode, ist die Zieladresse gerade, wird in den ARM Mode gewechselt.
Der Befehl BX LR ist der Standardbefehl, um Unterprogramme zu verlassen, bei denen die Rücksprungadresse noch im Linkregister steht (Return).

- BLX Label **Branch and Link with Exchange**

Der Befehl BLX Label wird ab ARM v5 genutzt in Programmen mit gemischtem 32-Bit/Thumb Code (Call).

- BLX Rn **Branch and Link with Exchange**

Dieser Befehl führt ebenfalls (optional) einen Modewechsel durch und speichert die nächste Adresse nach diesem Befehl im Linkregister (Call) (ab ARM v5).

# Unterprogrammaufruf mit Modewechsel

Beispiel:

```
        BL SUBR          @ Sprung zu SUBR (bleibt im ARM-Mode)
          …              @ Rücksprung hierher
          …
SUBR:     …              @ Eintrittspunkt der Subroutine
        BX LR            @ Return
                         @   (funktioniert in ARM+Thumb-Mode)
```

# Befehle zur Ablaufsteuerung

| | Sprungbefehle mit festem Offset | Sprungadresse in Register und Wechsel des Befehls-satzes in Abh. von Adresse |
|---|---|---|
| ohne Speicherung des PC in LR | **B Label** | **BX $R_d$** |
| mit Speicherung des PC in LR | **BL Label** | **BLX $R_d$** |
| mit Speicherung des PC in LR und Wechsel des Befehlssatzes | **BLX Label** | |

- Der Befehl *MOV PC, Rd* ist seit Einführung des Thumb Befehlssatzes, der einen direkten Zugriff auf den PC in diesem Befehl nicht mehr zuließ, nicht mehr empfohlen. Stattdessen wurden die Befehle BX und BLX eingeführt.
- Alle Sprungbefehle (inkl. Calls) können bedingt ausgeführt werden

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

## ARM-Befehle Fortsetzung…