Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

HTA-FR – Kurs Telekommunikation

Verfasser:
D. Gachet / HTA-FR - Telekommunikation

Embedded systems 1 und 2

Adressierungsmodi

Klasse T-2 // 2018-2019

Inhalt

- Wiederholung
- Verarbeitung der Daten
- Austausch der Daten mit dem Speicher
- Austausch der Daten mit den Coprozessoren

- **Architektur**
  - ARM9 ist ein RISC-Prozessor, der auf einer 32-Bit-Architektur basiert
  - Sein Befehlssatz ist mit Worten einer festen Länge von 32 Bit codiert
  - Alle arithmetischen und logischen Operationen werden auf 32-Bit-Worten ausgeführt

- **Funktionsweise**
  - ARM9 funktioniert nach dem Prinzip "Load and Store"
  - Für die Ausführung der Operationen mit den im Speicher abgelegten Daten muss der ARM9 die Daten zuerst aus dem Speicher in seine internen Register laden, anschliessend kann er die gewünschten Operationen mit diesen Daten ausführen und sie schliesslich wieder im Speicher ablegen.

Adressierungsmodi:

Die ARM-Prozessoren kennen 5 verschiedene Adressierungsmodi.
Diese lassen sich in drei grossen Kategorien zusammenfassen:

- **Verarbeitung der Daten**
  - Modus 1: "data-processing operands"
- **Austausch der Daten mit dem Speicher**
  - Modus 2: "Load and Store Word or Unsigned Byte"
  - Modus 3: "Miscellaneous Loads and Stores"
  - Modus 4: "Load and Store Multiple"
- **Austausch der Daten mit den Coprozessoren**
  - Modus 5: "Load and Store Coprozessor"

Referenz: Kapitel A5 des 01_ARM_Architecture_Reference_Manual.pdf

Verarbeitung der Daten

- **Die Verarbeitung der Daten kann nur mit 32-Bit-Daten erfolgen, die sich in den internen Registern R0 bis R15 des Prozessors befinden.**

- **Diese Verarbeitungen werden mithilfe der 3 Operanden in allgemeiner Form beschrieben:**

  <Operand_1> = <Operand_2> <Operation> <Operand_3>

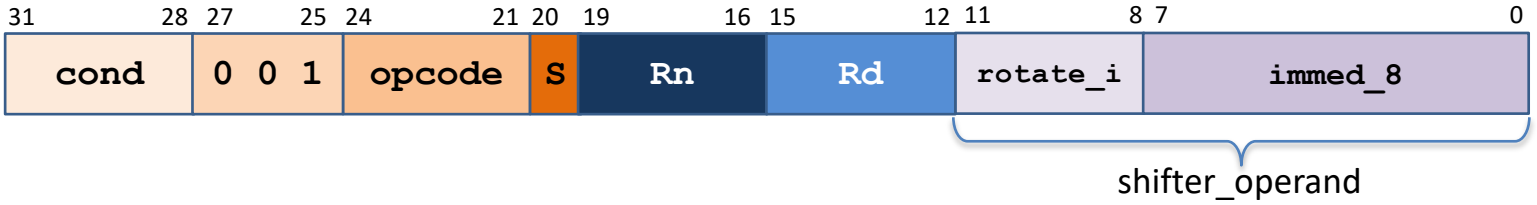
- **Der Adressierungsmodus 1, "Data-Processing Operands" wird für die Berechnung des Wertes des 3. Operanden, Verschiebungsoperand *"shifter_operand"* genannt, verwendet.**

  Syntax: <opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

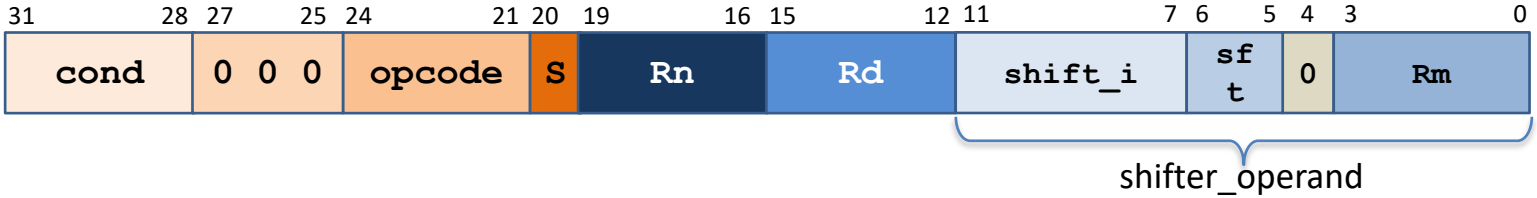
  - <opcode>: Operation (mov, add, sub, …)
  - <cond>: Ausführungsbedingung (optional)
  - {S}: Aktualisierung der Flags nach der Ausführung (optional)
  - <Rd>: 1. Operand ➔ Zielregister
  - <Rn>: 2. Operand
  - <shifter_operand>: 3. Operand

Verschiebungsoperand "Shifter Operand"

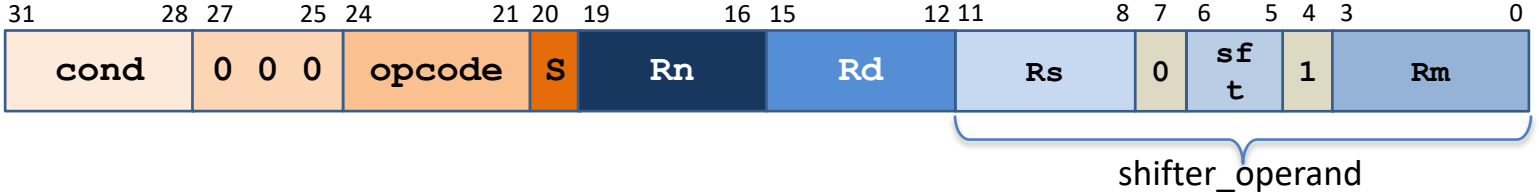
- **Der Verschiebungsoperand kann verschiedene Formen annehmen:**
  - Unmittelbar: #<immediate>



  - Register: <Rm>
  - Verschiebung/Rotation unmittelbarer Wert: <Rm>, LSL¦LSR¦ASR¦ROR #<shift_immediate>
  - Erweiterte Rotation: <Rm>, RRX

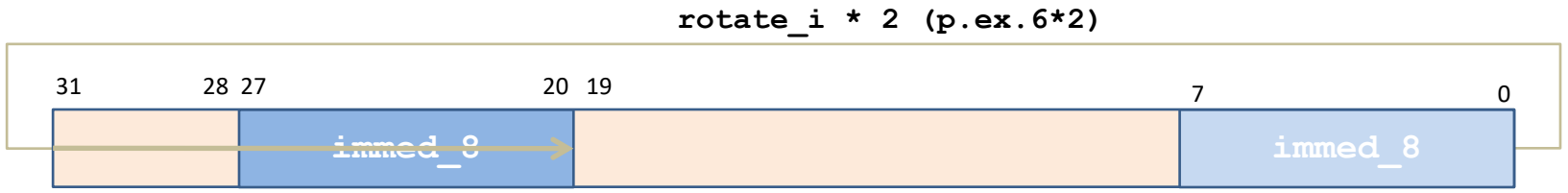


  - Verschiebung/Rotation durch Register: <Rm>, LSL¦LSR¦ASR¦ROR <Rs>



Verschiebungsoperand: Unmittelbar

- **Der Wert des 3. Operanden "Shifter Operand" wird durch Ausführung einer Rechtsrotation des unmittelbaren Wertes "immed_8" berechnet (im Befehl enthaltenes 8-Bit-Wort innerhalb eines 32-Bit-Wortes).**
- **Die Position des 8-Bit-Wortes wird erhalten, indem das im Befehl "rotate_i" enthaltene 4-Bit-Wort mit 2 multipliziert wird.**

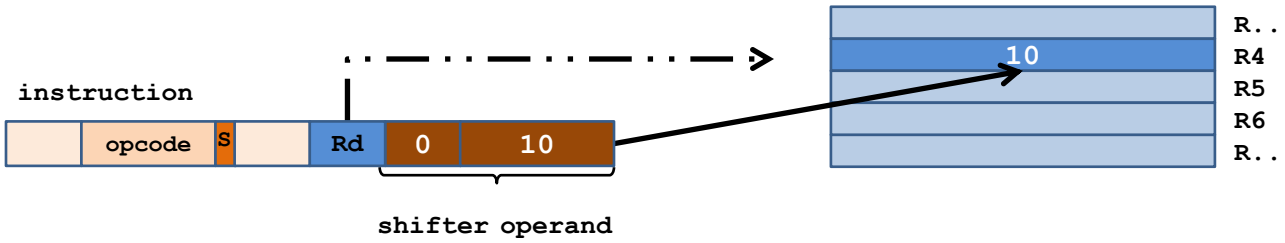


- **Einige mögliche Werte:**
  - 0xff, 0x104, 0xff0, 0xf000000f
- **Einige unmögliche Werte:**
  - 0x101, 0x102, 0xff1, 0xff04, 0xf000001f

Verschiebungsoperand: Immediate – Beispiele

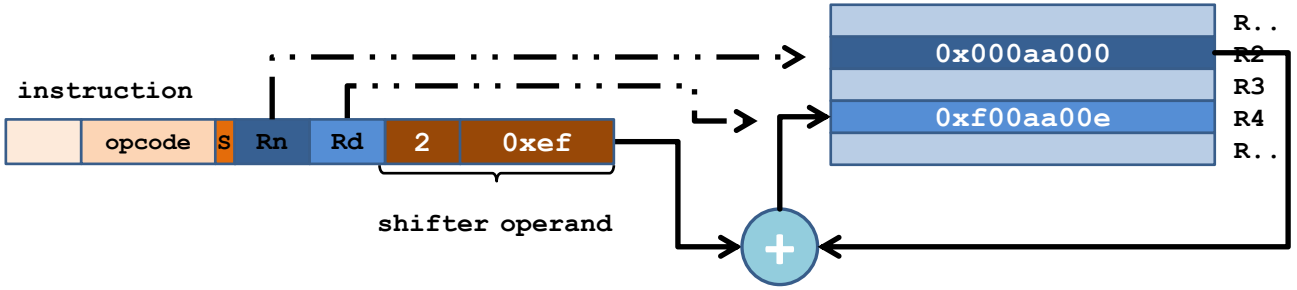
- **Beispiel:** `mov r4, #10`

  Schreibt den dezimalen Wert 10 in das Register R4.



- **Beispiel:** `add r4, r2, #0xf000000e`

  Schreibt das Ergebnis der Addition des Wertes im Register R2 mit dem hexadezimalen Wert 0xef nach einer Rotation um 4 Stellen nach rechts in das Register 4.

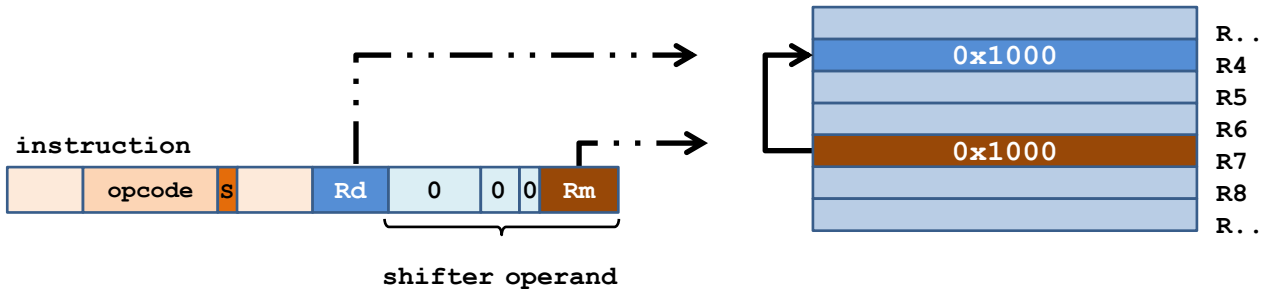


Verschiebungsoperand: Register

- **Der im Register <Rm> enthaltene Wert wird durch den Datenverarbeitungsbefehl direkt als Operand verwendet.**

- **Beispiel:** `mov r4, r7`

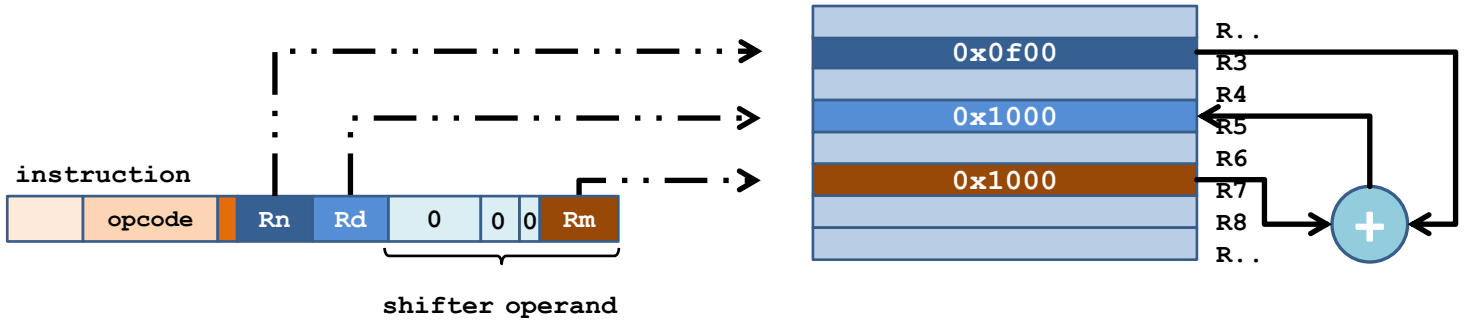
  Kopiert den Inhalt von Register R7 in das Register R4.



Verschiebungsoperand: Register (II)

- **Beispiel:** `add r5, r3, r7`

  Der Inhalt von Register R3 wird zum Inhalt von Register R7 addiert.
  Das Ergebnis wird in das Register R5 geschrieben

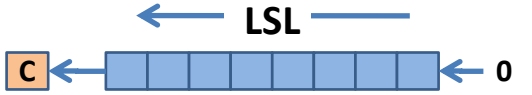


Verschiebungsoperand: Verschiebung und Rotation

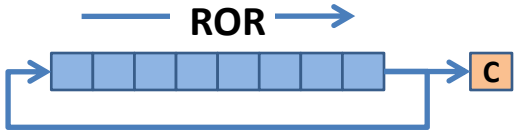
- **Der Wert des 3. Operanden wird durch Verschiebung oder Rotation des Wertes in einem der Prozessorregisters erhalten.**
- **Die 5 Typen für die Verschiebung und Rotation sind:**
  - ASR arithmetische Verschiebung nach rechts

  - LSR logische Verschiebung nach rechts

  - LSL logische Verschiebung nach links



  - ROR Rotation nach rechts



  - RRX erweiterte Rotation mit Carry nach rechts

- **Die Anzahl Verschiebungs- oder Rotationsbit kann wie folgt spezifiziert werden, entweder**
  - mit einem unmittelbaren Wert
  - oder mithilfe eines Registers

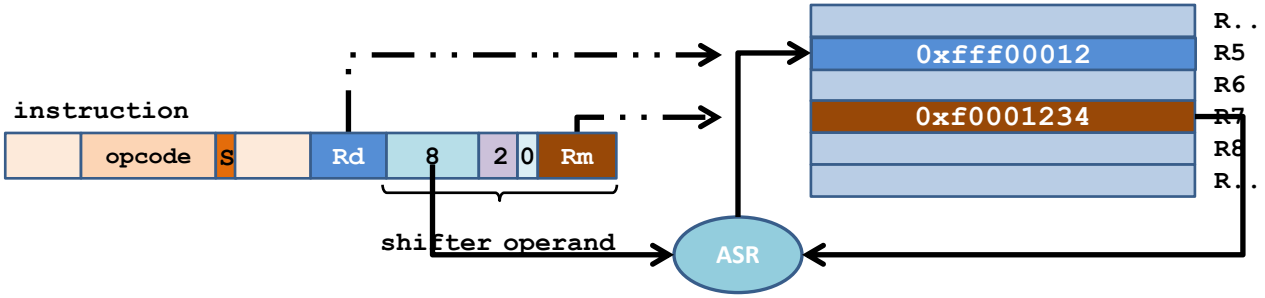
Verschiebungsoperand: Verschiebung und Rotation - ASR

- **Mit ASR wird der Verschiebungsoperand durch Verschieben des Wertes im Register <Rm> nach rechts gebildet (gleichbedeutend mit einer Division durch 2 mit Vorzeichen).**
- **Das höherwertige Bit (Rm[31]) wird anstelle des fehlenden Bit eingefügt.**

- **Das niederwertige, verschobene Bit (Rm[0] wird in das Carry geschrieben, sofern eine Aktualisierung der Flags gewünscht wird (S==1).**

- **Beispiel:** `mov r5, r7, ASR #8`

  Der Inhalt von Register R7 wird arithmetisch um 8 Bit nach rechts verschoben und das Ergebnis wird in das Register R5 geschrieben.



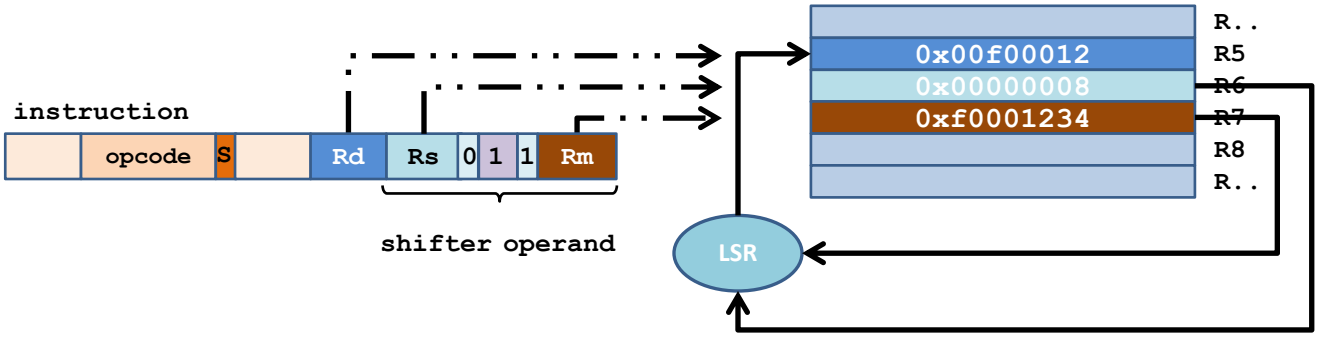
Verschiebungsoperand: Verschiebung und Rotation - LSR

- **Mit LSR wird der Verschiebungsoperand durch Verschieben des Wertes im Register <Rm> nach rechts gebildet (gleichbedeutend mit einer Division durch 2 ohne Vorzeichen) .**
- **Anstelle des fehlenden Bit wird eine 0 eingefügt.**

- **Das niederwertige, verschobene Bit (Rm[0] wird in das Carry geschrieben, sofern eine Aktualisierung der Flags gewünscht wird (S==1).**

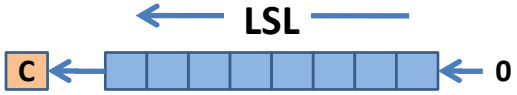
- **Beispiel:** `mov r5, r7, LSR r6`

Der Inhalt von Register R7 wird um die Anzahl Bit nach rechts verschoben, die im Register R6 definiert ist, und das das Ergebnis wird in das Register R5 geschrieben.



Verschiebungsoperand: Verschiebung und Rotation - LSL

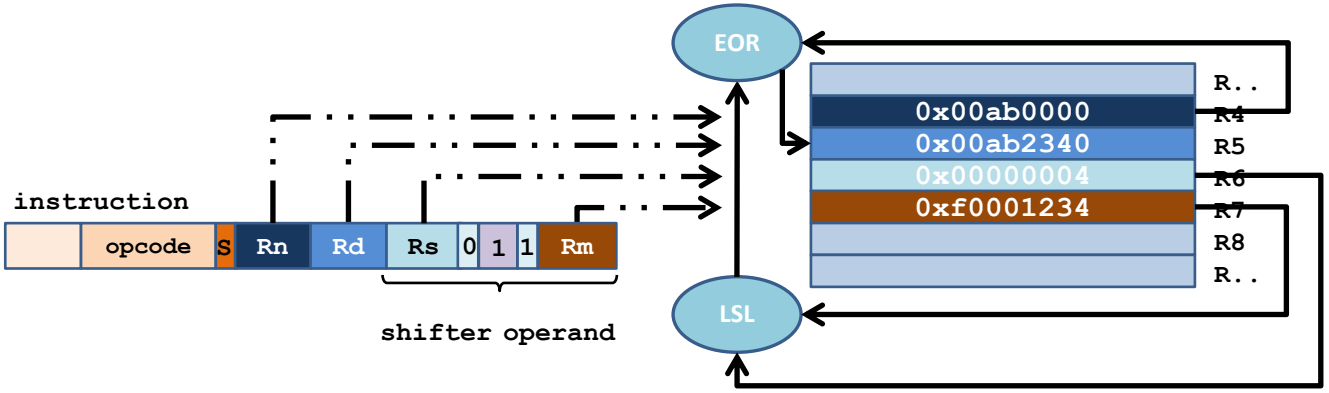
- **Mit LSL wird der Verschiebungsoperand durch Verschieben des Wertes im Register <Rm> nach links gebildet (gleichbedeutend mit einer Multiplikation mit 2).**
- **Anstelle des fehlenden Bit wird eine 0 eingefügt.**



- **Das letzte höherwertige, verschobene Bit (Rm[31] wird in das Carry geschrieben, sofern eine Aktualisierung der Flags gewünscht wird (S==1).**

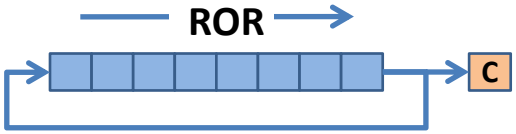
- **Beispiel:** `orr r5, r4, r7, LSL r6`

Der Inhalt von Register R7 wird um die Anzahl Bit nach links verschoben, die im Register R6 definiert ist, und anschliessend der Inhalt von Register R4 addiert (logische bitweise Addition). Das Ergebnis wird anschliessend in das Register R5 geschrieben



Verschiebungsoperand: Verschiebung und Rotation - ROR

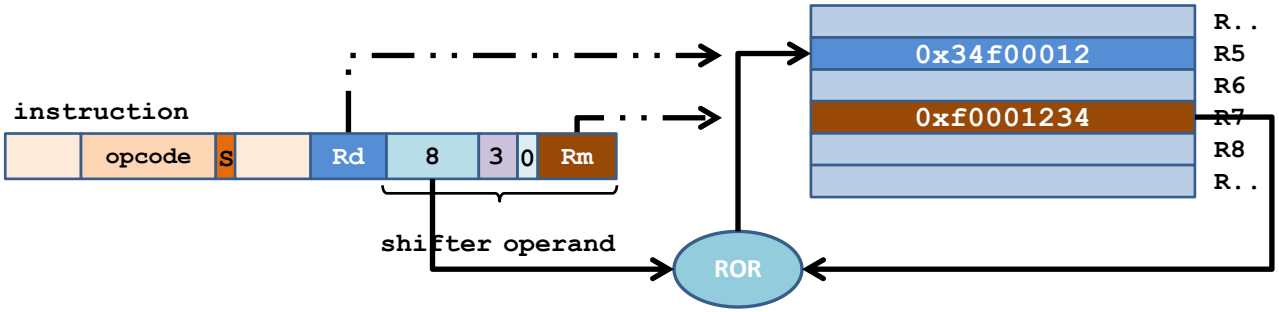
- **Mit ROR wird der Verschiebungsoperand durch das Ausführen einer Rotation des Inhalts von Register <Rm> nach rechts gebildet.**
- **Das niederwertige Bit (Rm[0]) wird anstelle des höherwertigen Bit (Rm[31]) eingefügt.**



- **Das niederwertige, verschobene Bit (Rm[0] wird in das Carry geschrieben, sofern eine Aktualisierung der Flags gewünscht wird (S==1).**

- **Beispiel:** `mov r5, r7, ROR #8`

  Der Inhalt von Register R7 wird um 8 Bit nach rechts rotiert und das Ergebnis wird in das Register R5 geschrieben.



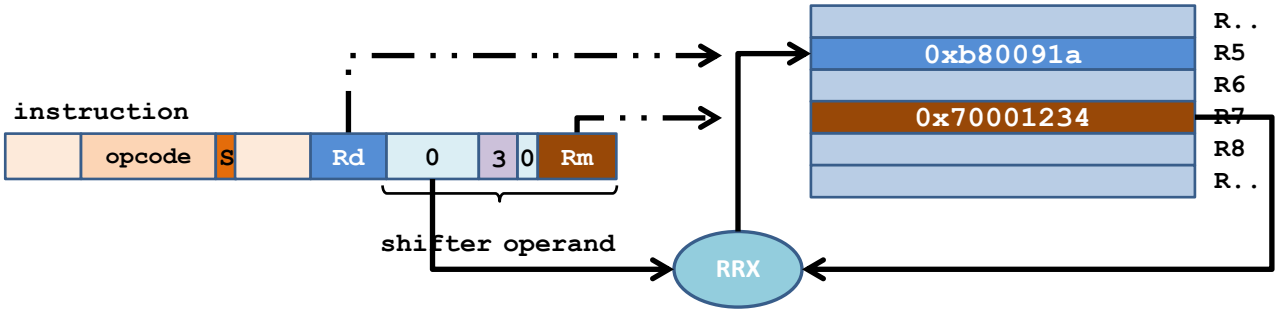
Verschiebungsoperand: Verschiebung und Rotation - ROX

- **Mit ROX wird der Verschiebungsoperand durch das Ausführen einer Rotation des Inhalts von Register <Rm> um 1 Bit nach rechts gebildet.**
- **Das Carry wird anstelle des höherwertigen Bit (Rm[31]) eingefügt.**

- **Das niederwertige Bit (Rm[0] wird in das Carry geschrieben, sofern eine Aktualisierung der Flags gewünscht wird (S==1).**

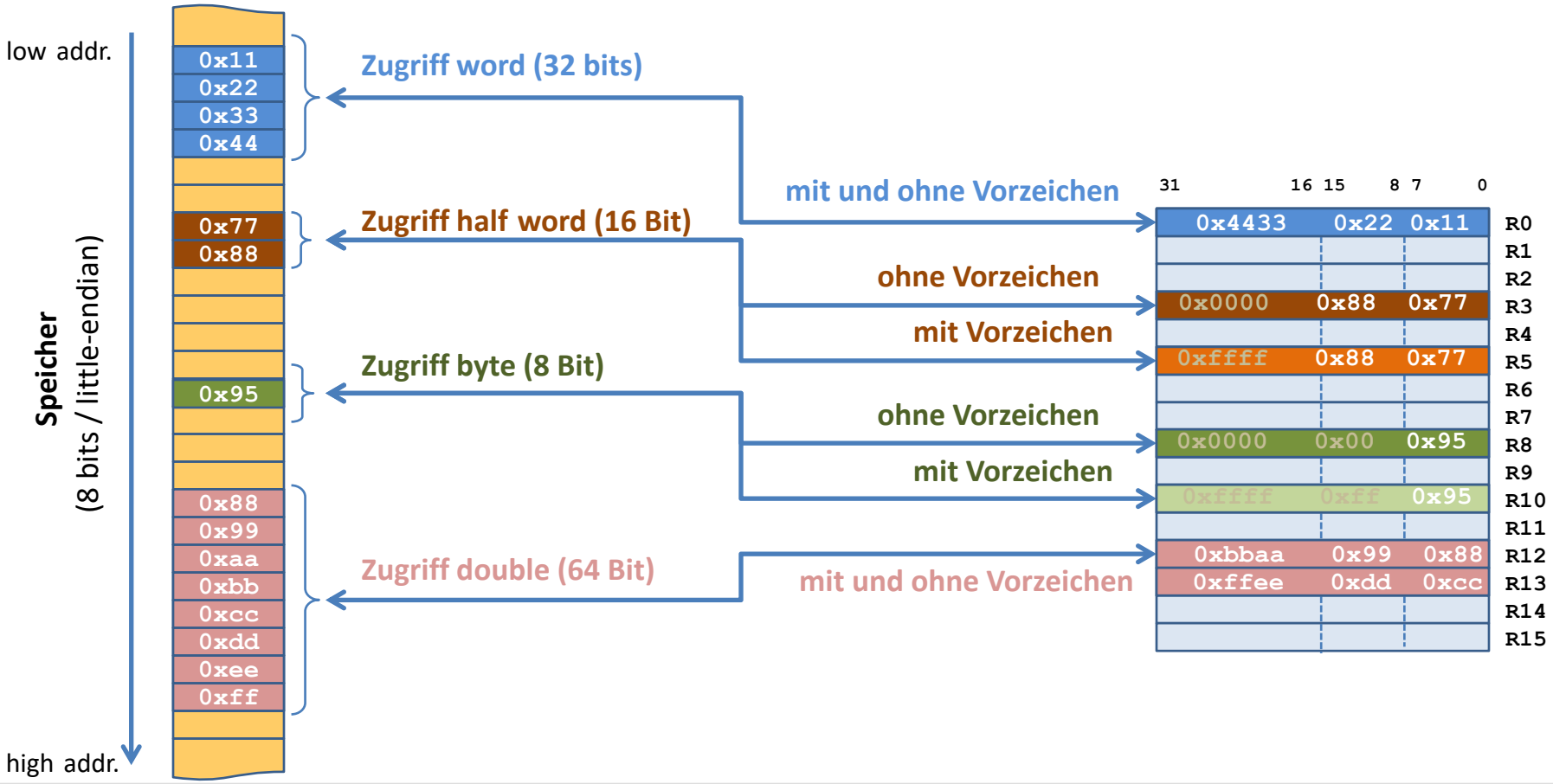
- **Beispiel:** `movs r5, r7, RRX ; C==1`

  Der Inhalt des Registers R7 wird über Carry um 1 Bit nach rechts rotiert. Das Carry wird in das Bit 31 eingefügt. Bit 0 wird in das Carry eingefügt. Das Ergebnis wird in das Register R5 geschrieben



Austausch der Daten mit dem Speicher

Die ARM-Prozessoren arbeiten nach dem Prinzip "Load and Store".
Dieses Prinzip impliziert, dass die Daten für die Verarbeitung zwingend in die Register des Prozessors geladen *("load")* sein müssen und dann, nach der Verarbeitung, erneut im Hauptspeicher gespeichert *("store")* werden können.



Austausch der Daten mit dem Speicher (II)

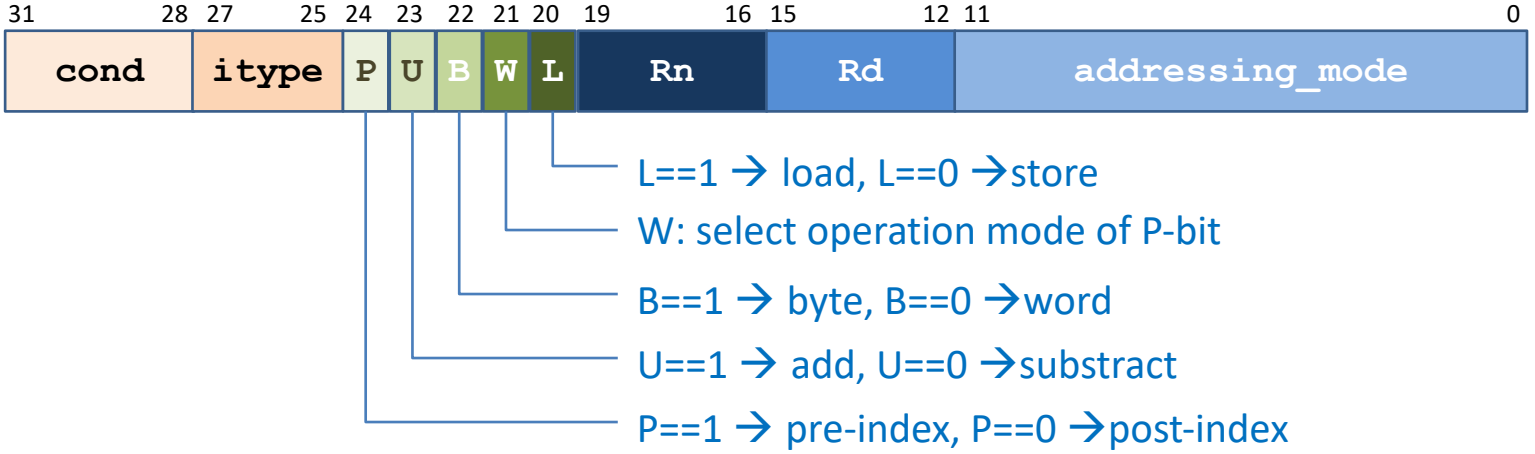
- **Befehle:**

  Syntax: LDR¦STR{<cond>}{H|B¦D¦SH¦SB} <Rd>, <addressing_mode>

  - LDR¦STR: Befehl (load oder store)
  - <cond>: Ausführungsbedingung (optional)
  - <Rd>: Ziel-/Quellregister
  - {H¦B¦D¦SH¦SB}: Grösse und Vorzeichen des Datenelements, vorgegeben word (32 Bit)

    H: halfword, B: byte, D: double
    SH: signed halfword, SB: signed byte
  - < addr_mode >: Adressierungsmodus

- **Codierung der Operanden im Befehl:**

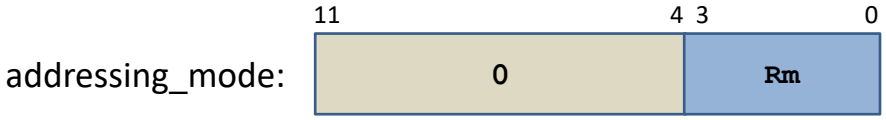


Operand des Adressierungsmodus "Addressing Mode"

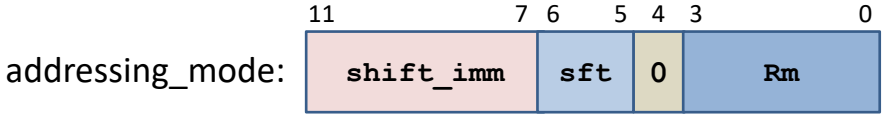
- **Für 32-Bit- und 8-Bit-Worte ohne Vorzeichen kann der Operand des Adressierungsmodus (Offset/Index) die drei nachfolgenden Formen annehmen:**

  - Unmittelbar: ldr{b}/str{b} <Rd>, [<Rn>, #+/-<offset_12>]
    vorindexiert: ldr{b}/str{b} <Rd>, [<Rn>, #+/-<offset_12>]!
    nachindexiert: ldr{b}/str{b} <Rd>, [<Rn>], #+/-<offset_12>

  - Durch Register: ldr{b}/str{b} <Rd>, [<Rn>, +/-<Rm>]
    vorindexiert: ldr{b}/str{b} <Rd>, [<Rn>, +/-<Rm>]!
    nachindexiert: ldr{b}/str{b} <Rd>, [<Rn>], +/-<Rm>



  - Durch Register und Verschiebung: ldr{b}/str{b} <Rd>, [<Rn>, +/-<Rm>, <shift> #<shift_imm>]
    vorindexiert: ldr{b}/str{b} <Rd>, [<Rn>, +/-<Rm>, <shift> #<shift_imm>]!
    nachindexiert: ldr{b}/str{b} <Rd>, [<Rn>], +/-<Rm>, <shift> #<shift_imm>



Operand des Adressierungsmodus "Addressing Mode" (II)

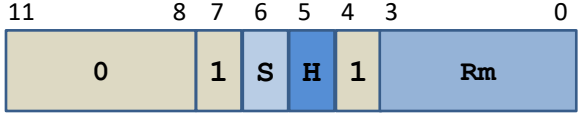
- **Für 16-Bit-Worte mit/ohne Vorzeichen, 8-Bit-Worte mit Vorzeichen und 64-Bit-Worte kann der Operand des Adressierungsmodus (Offset/Index) die folgenden Formen annehmen:**

  - Unmittelbar: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>, #+/-<immediate_8>]`
    vorindexiert: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>, #+/-<immediate_8>]!`
    nachindexiert: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>], #+/-<immediate_8>`

addressing_mode:

  - Durch Register: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>, +/-<Rm>]`
    vorindexiert: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>, +/-<Rm>]!`
    nachindexiert: `ldr/str h¦sh¦sb¦d <Rd>, [<Rn>], +/-<Rm>`

addressing_mode:

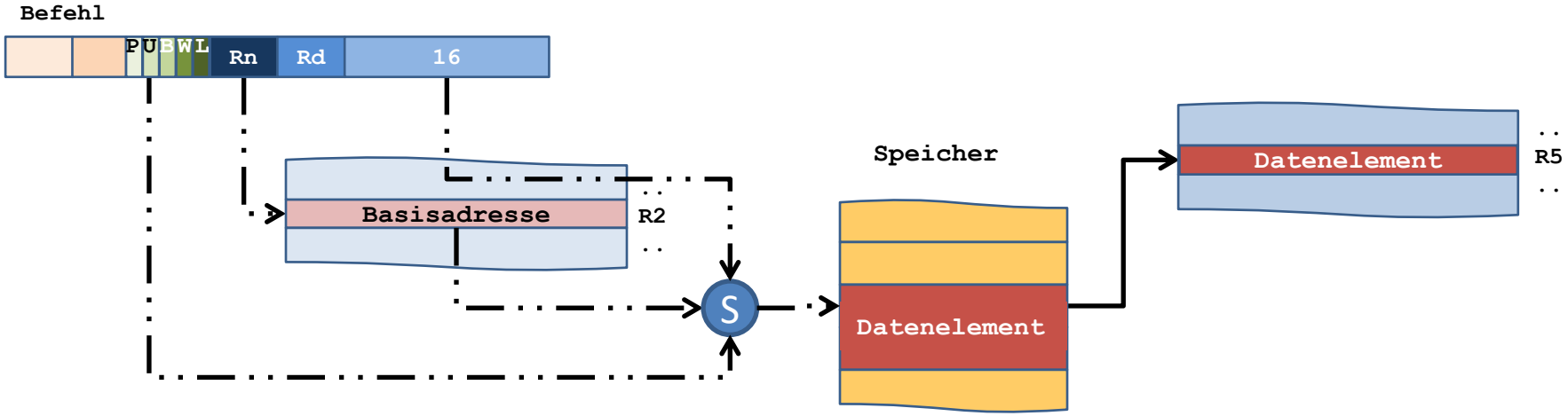


Adressierungsmodus: Immediate offset/index

- **Die tatsächliche Adresse des Datenelements im Speicher wird berechnet, indem der im Befehl enthaltene Offset-/Indexwert zum Wert im Register <Rn> addiert oder von ihm subtrahiert wird. Mögliche Werte für:**
  - 32- und 8-Bit-Worte ohne Vorzeichen: 0 .. 4095.
  - 64-, 16- und 8-Bit-Worte mit Vorzeichen: 0 .. 255
- **Nach der Berechnung der tatsächlichen Adresse wird das an dieser Adresse im Speicher abgelegte Datenelement in das Register <Rd> geschrieben oder umgekehrt.**

- **Beispiel:** `ldr r5, [r2, #16]`

  Legt das an der Speicheradresse R2 + 16_{10} enthaltene Datenelement (32 Bit) im Register R5 ab.

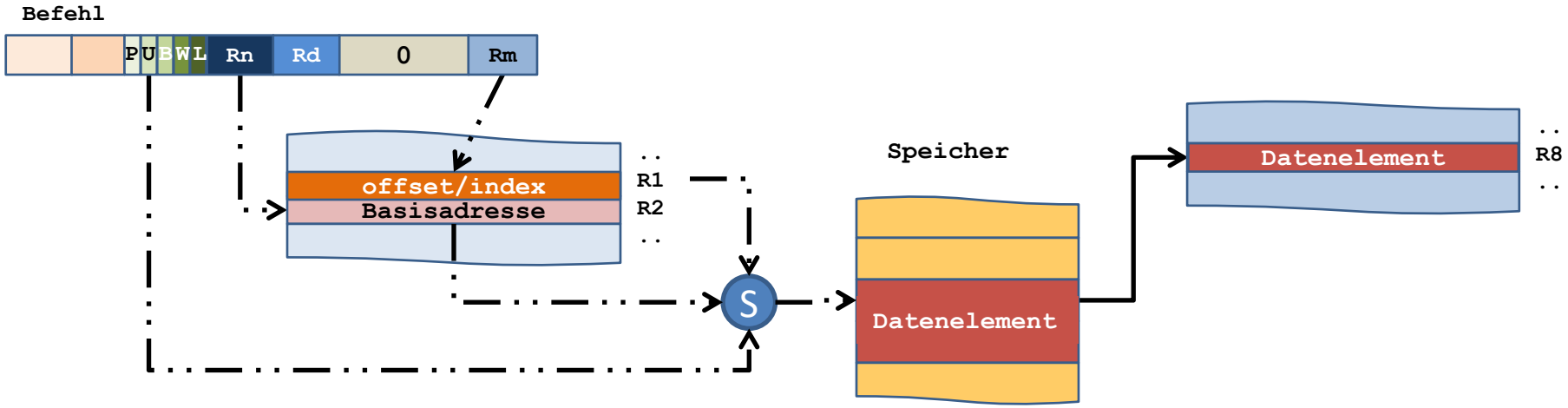


Adressierungsmodus: Register offset/index

- **Die tatsächliche Adresse des Datenelements im Speicher wird berechnet, indem der im Register <Rm> enthaltene Offset-/Indexwert zum Wert im Register <Rn> addiert oder von ihm subtrahiert wird.**
- **Nach der Berechnung der tatsächlichen Adresse wird das an dieser Adresse im Speicher abgelegte Datenelement in das Register <Rd> geschrieben oder umgekehrt.**

- **Beispiel:** `ldrb r8, [r2, -r1]`

  Legt das an der Speicheradresse R2 - R1 enthaltene Datenelement (8 Bit) im Register R8 ab. Die 24 höherwertigen Bit werden auf 0 gesetzt.



Adressierungsmodus: Scaled register offset/index

- **Die tatsächliche Adresse des Datenelements im Speicher wird berechnet, indem der im Register <Rm> enthaltene Offset-/Indexwert nach der Verschiebung oder Rotation um die Anzahl Bit, die durch "shift_imm" spezifiziert und im Befehl codiert ist, zum Wert im Register <Rn> addiert oder von ihm subtrahiert wird.**

- **Die 5 Varianten** *(identisch mit dem Verschiebungsoperand)* **sind:**
  - [<Rn>, +/-<Rm>, LSL #<shift_imm>] // shift_imm: 0..31
  - [<Rn>, +/-<Rm>, LSR #<shift_imm>] // shift_imm: 1..32
  - [<Rn>, +/-<Rm>, ASR #<shift_imm>] // shift_imm: 1..32
  - [<Rn>, +/-<Rm>, ROR #<shift_imm>] // shift_imm: 1..31
  - [<Rn>, +/-<Rm>, RRX]

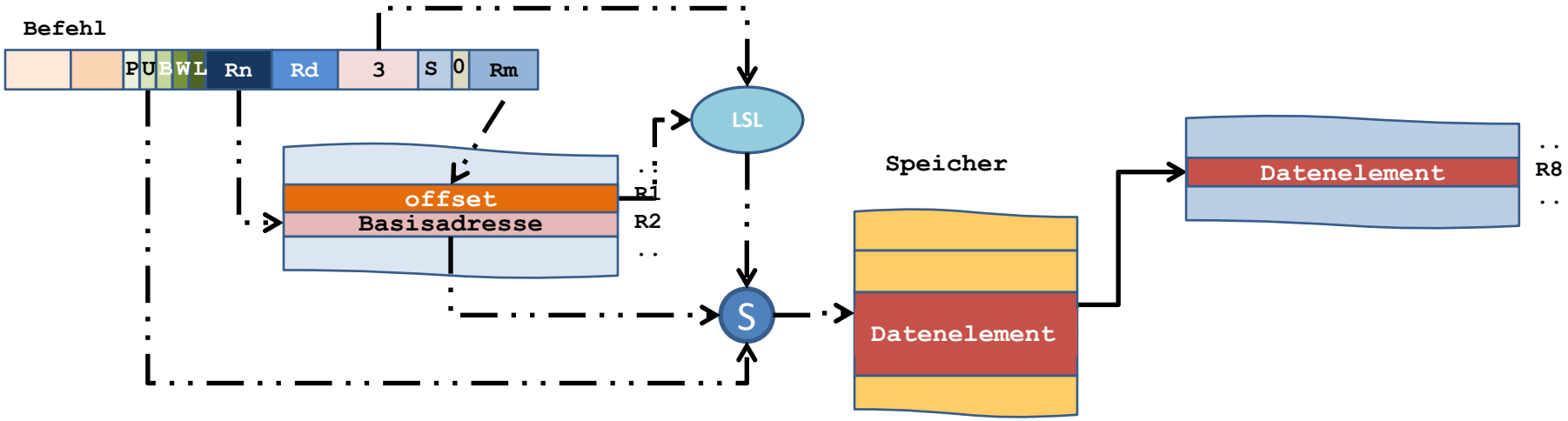
- **Nach der Berechnung der tatsächlichen Adresse wird das an dieser Adresse im Speicher abgelegte Datenelement in das Register <Rd> geschrieben oder umgekehrt.**

- **Dieser Adressierungsmodus ist nur für 32- oder 8-Bit-Daten ohne Vorzeichen möglich!**

Adressierungsmodus: Scaled register offset/index (II)

- **Beispiel:** `ldrb r8, [r2, +r1, LSL #3]`

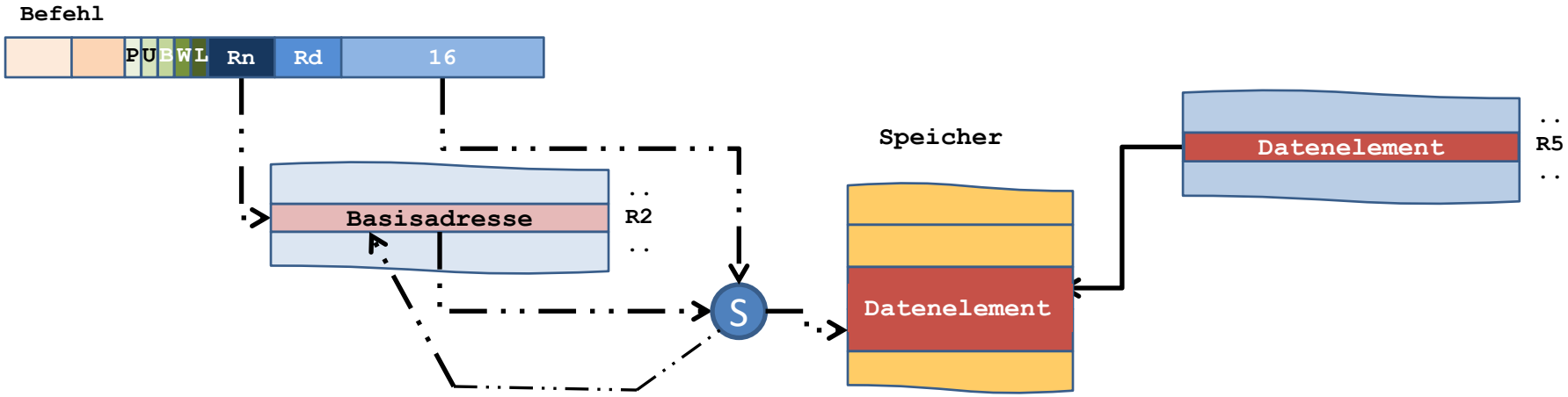
  Legt das an der Speicheradresse R2 + (R1 << 3) enthaltene Datenelement (8 Bit) im Register R8 ab. Die 24 höherwertigen Bit werden auf 0 gesetzt.



Adressierungsmodus: Pre-indexed

- **Der Adressierungsmodus "vorindexiert" erlaubt, das Ergebnis der Berechnung der tatsächlichen Adresse des Datenelements im Speicher entsprechend einer der 3 oben erwähnten Methoden im Register <Rn> zu speichern.**
- **Auf das Datenelement wird erst nach der Berechnung der tatsächlichen Adresse zugegriffen.**
- **Dieser Modus erlaubt, sehr einfach Iterationsschleifen zu implementieren.**

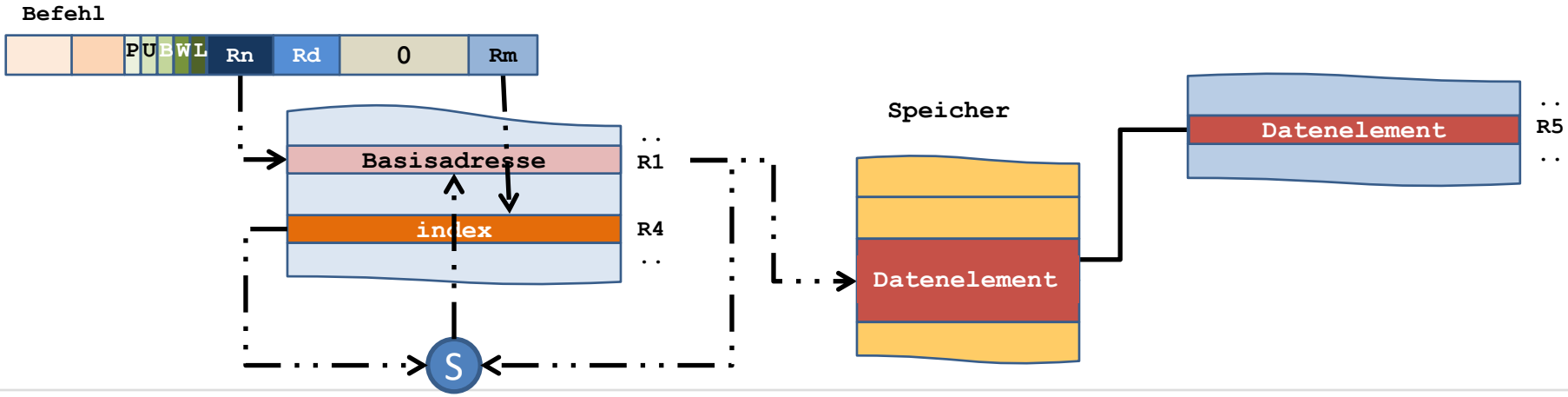
- **Beispiel:** `strh r5, [r2, #10]!`
  - Berechnet die tatsächliche Adresse des Datenelements im Speicher, indem der Inhalt von R2 zu 10_{10} addiert und das Ergebnis im Register R2 abgelegt wird.
  - Das im Register R5 (16 niederwertige Bit) enthaltene Datenelement (16 Bit) wird anschliessend an der Speicheradresse abgelegt, die in R2 enthalten ist.



Adressierungsmodus: Post-indexed

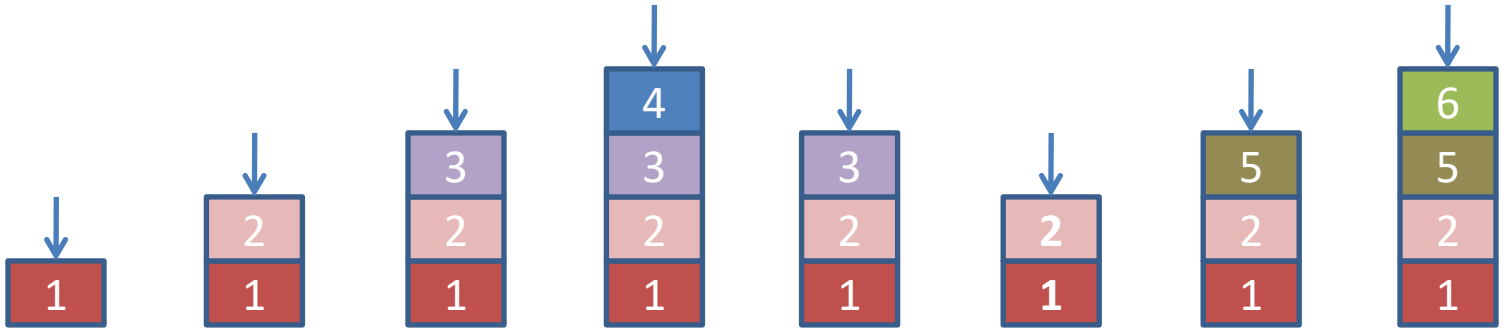
- **Der Adressierungsmodus "nachindexiert" erlaubt, das Ergebnis der Berechnung der tatsächlichen Adresse des Datenelements im Speicher entsprechend einer der 3 ersten Adressierungsmethoden im Register <Rn> zu speichern.**
- **Die Berechnung der neuen tatsächlichen Adresse erfolgt erst nach dem Zugriff auf das im Speicher abgelegte Datenelement.**
- **Dieser Modus erlaubt, sehr einfach Iterationsschleifen zu implementieren.**

- **Beispiel:** `strh r5, [r1], r4`
  - Das im Register R5 (16 niederwertige Bit) enthaltene Datenelement (16 Bit) wird an der Speicheradresse abgelegt, die in R1 enthalten ist.
  - Anschliessend wird die tatsächliche Adresse des Datenelements im Speicher berechnet, indem der Inhalt von R4 zu R1 addiert und das Ergebnis im Register R1 abgelegt wird.



Mehrfachaustausch von Daten

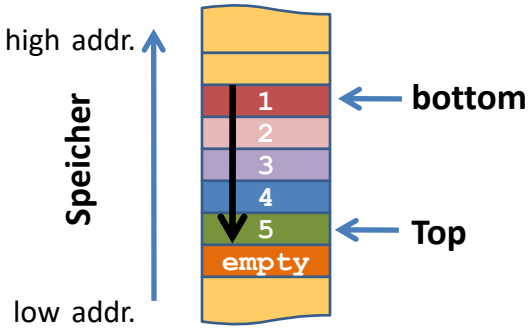
- Die Sicherung und Wiederherstellung eines Teils der oder aller Register des Prozessors ist eine ziemlich häufige Aufgabe.
- Diese Operationen sind beim Aufrufen einer Subroutine sehr interessant, um den Zustand der Register der aufrufenden Routine zu sichern. Sie werden speziell durch hoch entwickelte Programmiersprachen benutzt. Man spricht dann von einem Stapel ("stack").
- Der Stapel ("stack") ist lediglich eine lineare Liste von Elementen, der beim Zugriff Elemente hinzugefügt oder entnommen werden.



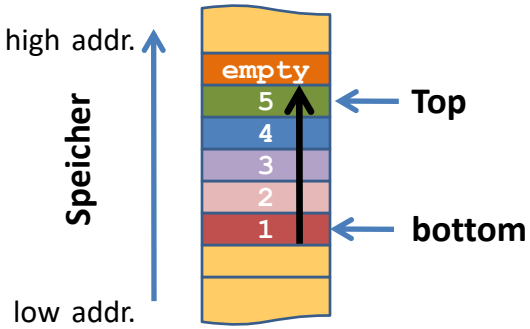
Darstellung in der Form eines Stapels (stack)

Mehrfachaustausch von Daten (II)

- **Die Daten können auf diesen Stapeln (stacks) auf zwei verschiedene Arten abgelegt werden:**
  - Absteigender Stapel (descending stack):
    Bei diesem Modell wächst der Stapel im Speicher nach unten, d. h. die Speicheradressen werden dekrementiert.

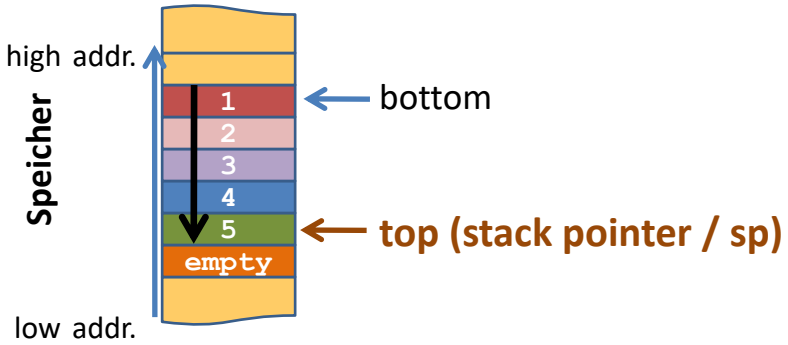


  - Aufsteigender Stapel (ascending stack):
    Bei diesem Modell wächst der Stapel im Speicher nach oben, d. h. die Speicheradressen werden inkrementiert.

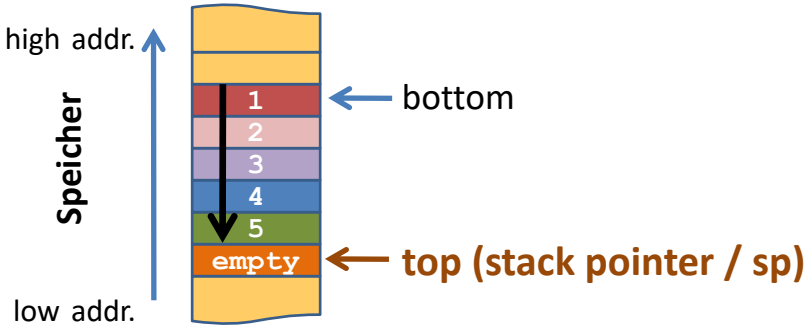


Mehrfachaustausch von Daten (III)

- **Die ARM-Prozessoren empfehlen zwei Stapeltypen:**
  - Voller Stapel (full stack):
    Der Stapelzeiger (Stackpointer) (top) zeigt die Position des letzten gespeicherten Wertes an.



  - Leerer Stapel (empty stack):
    Der Stapelzeiger (Stackpointer) (top) zeigt die Position des ersten unbenutzten Speicherplatzes an.



Mehrfachaustausch von Daten (IV)

- **Auf der Basis dieser verschiedenen Modelle bieten die ARM-Prozessoren einen Befehlssatz, der die Ausführung dieser Operationen und den Austausch der Registerinhalte mit jenen des Speichers ermöglicht.**

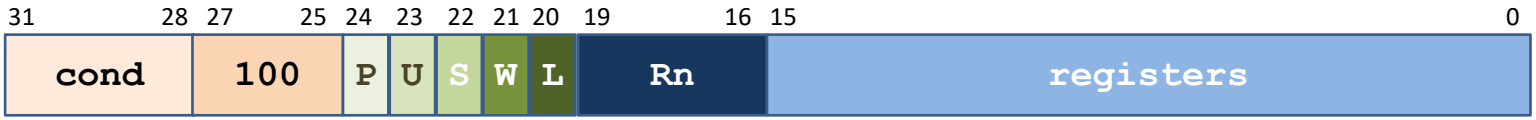
  Syntax: LDM¦STM{<cond>}<addr_mode> <Rn>{!}, <registers>{^}

  - LDM¦STM: Befehl (load multiple oder store multiple)
  - <cond>: Ausführungsbedingung (optional)
  - <Rn>: Register als Stapelzeiger (stack pointer / sp) benutzt
  - < addr_mode >: Adressierungsmodus
  - <registers>: Liste der zu sichernden/wieder herzustellenden Register
  - !: Aktualisierung des Registers <Rn> nach der Ausführung
  - ^: sichern/wieder herstellen der Benutzerregister

- **Der Inhalt des Registers mit der kleinsten Nummer wird an der niedrigsten Speicheradresse abgelegt, während jener des Registers mit der höchsten Nummer an der höchsten Speicheradresse abgelegt wird.**

Mehrfachaustausch von Daten (V)

- **Codierung der Operanden im Befehl:**



- **Adressierungsmodus in generischer Form:**
  - **IA: increment after** *(Sichern der Register in einer Struktur)*
  - IB: increment before
  - DA: decrement after
  - DB: decrement before

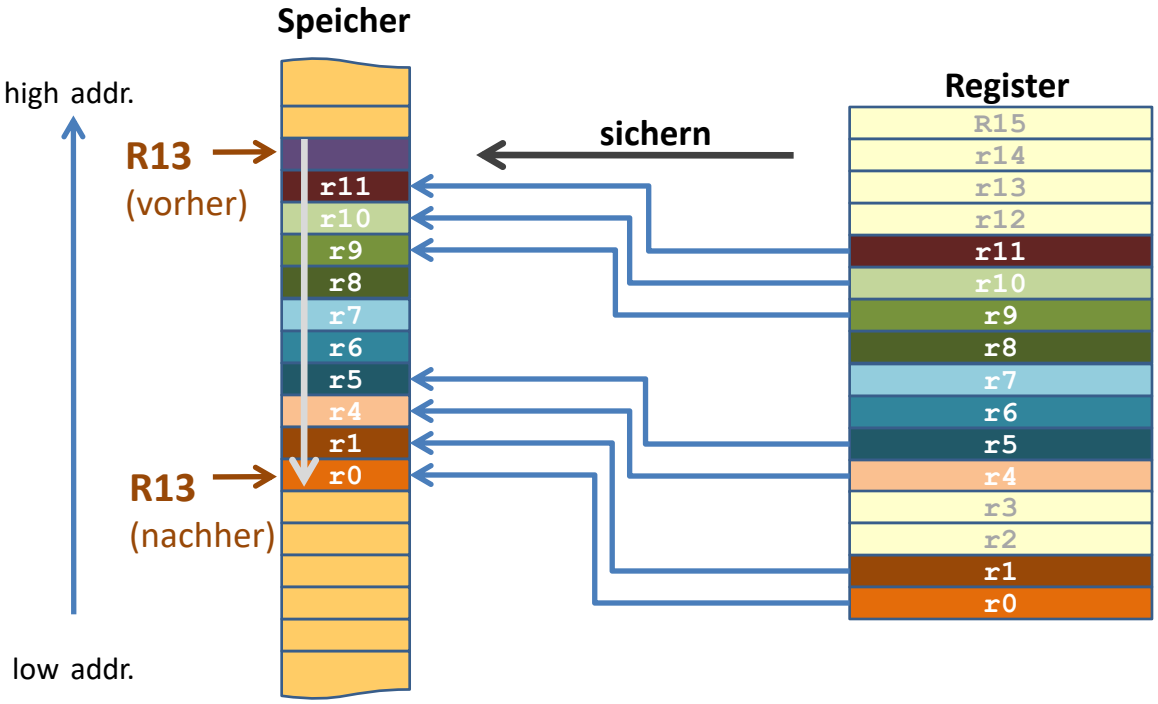
- **Adressierungsmodus für die Stapelverarbeitung (stack)**
  - FA: full ascending
  - **FD: full descending** *(C/C++ stack convention)*
  - EA: empty ascending
  - ED: empty descending

Mehrfachaustausch der Daten: Sichern der Register

- **Beispiel:** `stmfd r13!, {r0,r1,r4-r11}`

  Der Inhalt der Register R0, R1 und R4 bis R11 wird im Speicher an der im Register r13 (stack pointer) spezifizierten Adresse gesichert. R13 wird nach der Ausführung der Operation aktualisiert (dekrementiert).

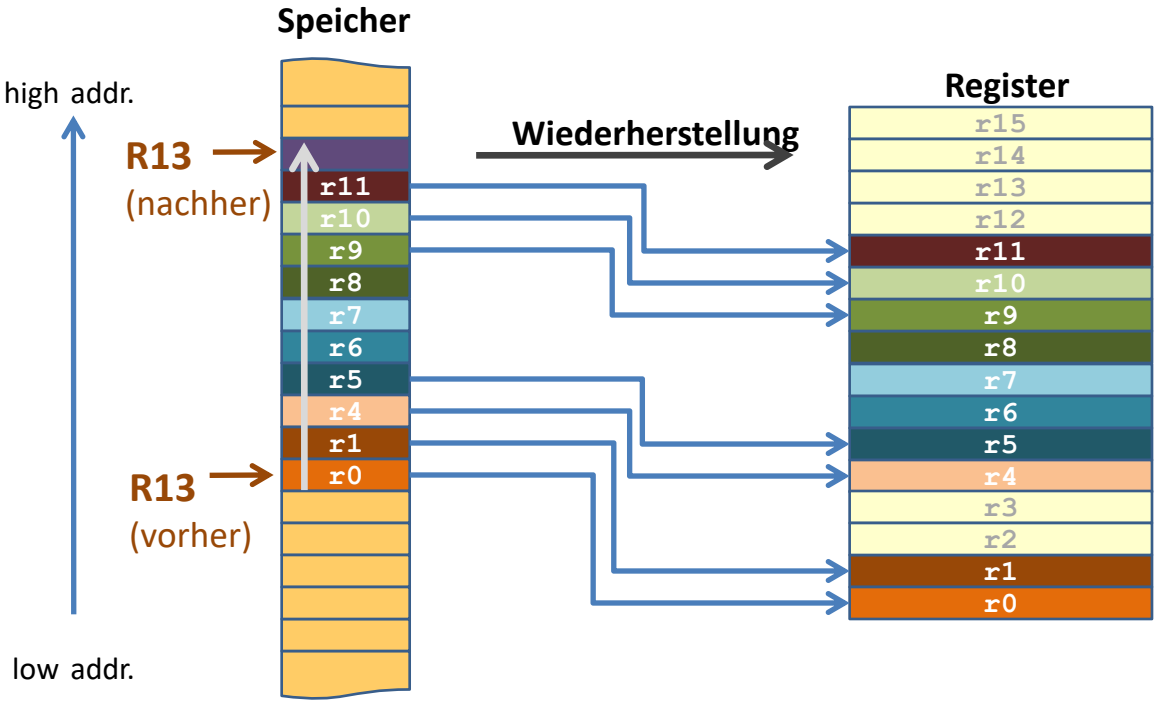
  Die verwendete Methode lautet: full descending stack



- **Beispiel:** `ldmfd r13!, {r0,r1,r4-r11}`

  Der Inhalt der Register R0, R1 und R4 bis R11 wird aus dem Speicher ab der im Register r13 (stack pointer) spezifizierten Adresse wieder hergestellt. R13 wird nach der Ausführung der Operation aktualisiert (inkrementiert).

  Die verwendete Methode lautet: full descending stack



Austausch der Daten mit den Coprozessoren

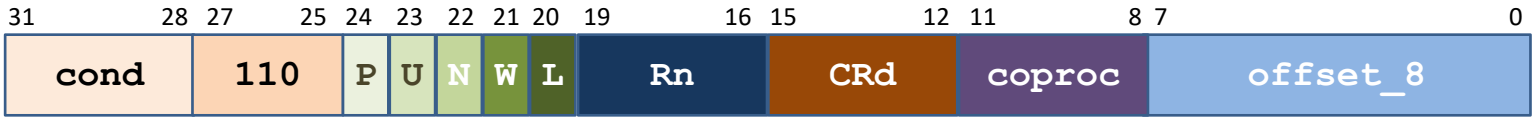
- **Die ARM-Prozessoren sind für die Ausführung spezieller Aufgaben, z. B. Speicherverwaltung (MMU), Cache usw. mit Coprozessoren ausgestattet.**
- **Für den Austausch von Daten mit den Coprozessoren implementiert der ARM9 einen Befehlssatz mit der folgenden Syntax:**

  Syntax: <opcode>{<cond>}{L} <coproc>, <CRd>, <addr_mode>

  - <opcode>: Operation (cdp, ldc, mcr, …)
  - <cond>: Ausführungsbedingung (optional)
  - {L}: load / store
  - < coproc >: Name des Coprozessors
  - < CRd >: Register des Coprozessors
  - < addr_mode >: Adressierungsmodus

Austausch der Daten mit den Coprozessoren (I)

- **Codierung der Operanden im Befehl:**



- **Die folgenden 4 Adressierungsmodi werden durch den Prozessor implementiert:**
  - Unmittelbar: [<Rn>, #+-<offset_8>]
  - Unmittelbar vorindexiert: [<Rn>, #+-<offset_8>]!
  - Unmittelbar nachindexiert: [<Rn>], #+-<offset_8>
  - Durch Option (nicht indexiert): [<Rn>], #<option>