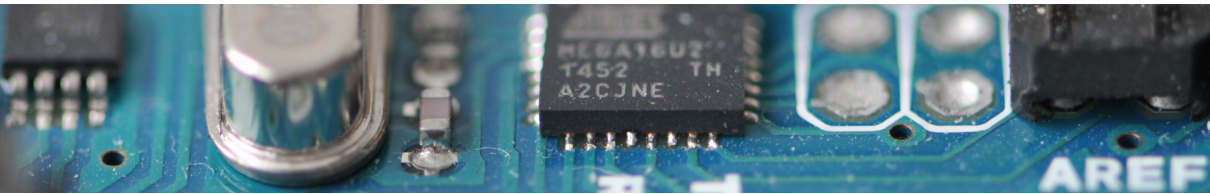
universität
innsbruck



Rechnerarchitektur

Befehlssatzarchitektur II

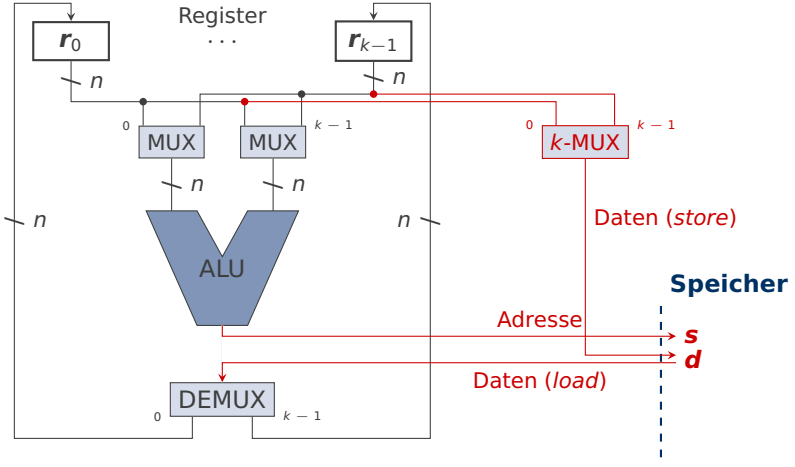
Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2020/21 · 2. Dezember 2020

Gliederung heute

1. **Speicherzugriff**
2. Division und Zahlenausgabe in Assembler
3. Stapelorganisation und Funktionsaufrufe

Schaltskizze eines Mikroprozessors

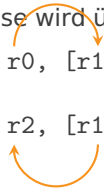


Darstellung ohne Statusregister bzw. Flags, *Load-Store*-Architektur ohne Instruktionsdekodierung

Speicherzugriff

Mnemonics			Kommentar
LDR	STR	SWP	Lese/schreibe/tausche 32-Bit-Wort
LDRB	STRB	SWPB	Lese/schreibe/tausche Byte
LDRH	STRH		Lese/schreibe Halbwort (16 Bit)
LDRSB			Lese Byte mit Vorzeichenerweiterung
LDRSH			Lese Halbwort mit Vorzeichenerweiterung

Die Adresse wird über ein **Basisregister** plus **Offset** angegeben:



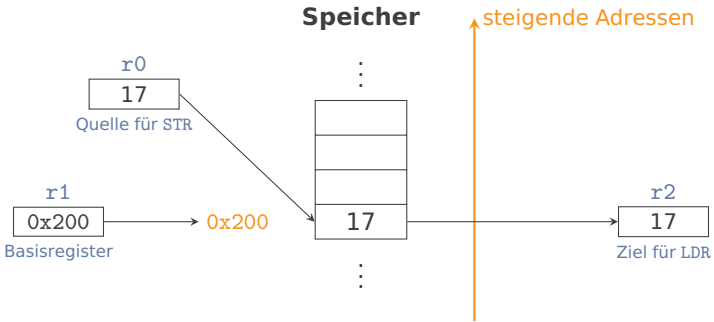
```
STR   r0, [r1]          ; Inhalt von r0 an Adresse speichern,
                        ; die in r1 steht.
LDR   r2, [r1,#-12]     ; Speicherinhalt an der Adresse (r1−12)
                        ; nach r2 laden.
```

Bedingungen sind möglich und werden zwischen Stamm-Mnemonic und Größensuffix eingeschoben, z. B. LDREQB.

Adressierungsarten

Angabe der Speicheradresse über **Basisregister**

```
STR   r0, [r1]    ; Inhalt von r0 an Adresse speichern, die in r1 steht.
LDR   r2, [r1]    ; Speicherinhalt an der Adresse r1 nach r2 laden.
```

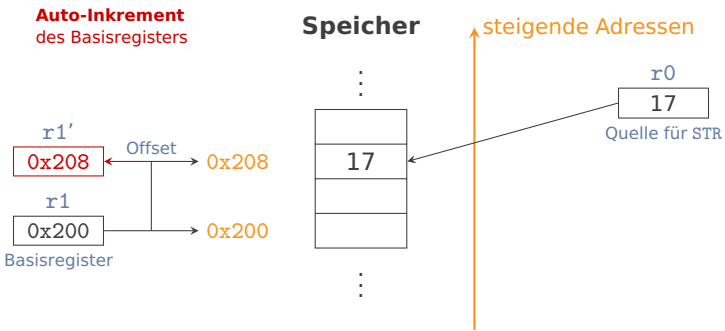


ARM unterstützt ausschließlich **indirekte** Adressierung.

Adressierungsarten (Forts.)

Angabe der Speicheradresse über **Basisregister** und **Offset**

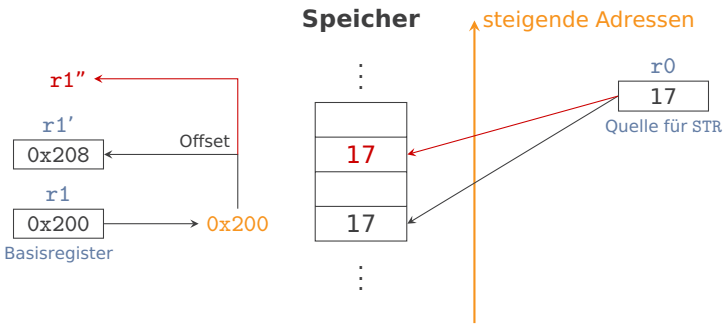
```
STR   r0, [r1, #8]!    ; Immediate (12 Bit plus Vorzeichen)
```



Adressierungsarten (Forts.)

Angabe der Speicheradresse über **Basisregister** und **Offset**

```
STR   r0, [r1], #8            ; "Post-indexed"-Adressierung
STR   r0, [r1], r2, LSL #3    ; mit Register (äquivalent falls r2 = 1)
```



Beispiele

Zugriff auf das *k*-te Element eines **Arrays**, das aus 16 Byte langen Datenstrukturen besteht

```
      ; erwarte k in r2
LDR   r1, =beispiel+4
      ; r1 zeigt auf feld[0].ziel
LDR   r0, [r1, r2, LSL #4]
      ; Lesezugriff, r1 unverändert
```

Beispiel-Struktur in C

```c
struct beispiel_t {
  unsigned int quelle;
  unsigned int ziel;
  int          anzahl;
  int          pad;
       } feld[1024];
```

Beispiele (Forts.)

Kopieren von Speicherbereichen

```
                ; ggf. Rücksprungadresse in lr vorher sichern
                ; besser: Variante mit niedrigeren Registern schreiben

            LDR   r12, =quelle   ; erste zu kopierende Adresse
            LDR   r13, =ziel     ; erste Zieladresse
            LDR   r14, =len      ; Länge in Wörtern (> 0, sonst fatal)
copyloop:
            LDR   r0, [r12], #4 ; Auto-Inkrement, post-indexed
            STR   r0, [r13], #4 ; Auto-Inkrement, post-indexed
            SUBS  r14, r14, #1
            BNE   copyloop

                ; Sonderbehandlung nötig, wenn Daten nicht "aligned"
```

Geht es noch effizienter ?

Block Data Transfer

Mnemonic	Kommentar
`LDM`	lese 1–16 Register (*load multiple*)
`STM`	schreibe 1–16 Register (*store multiple*)

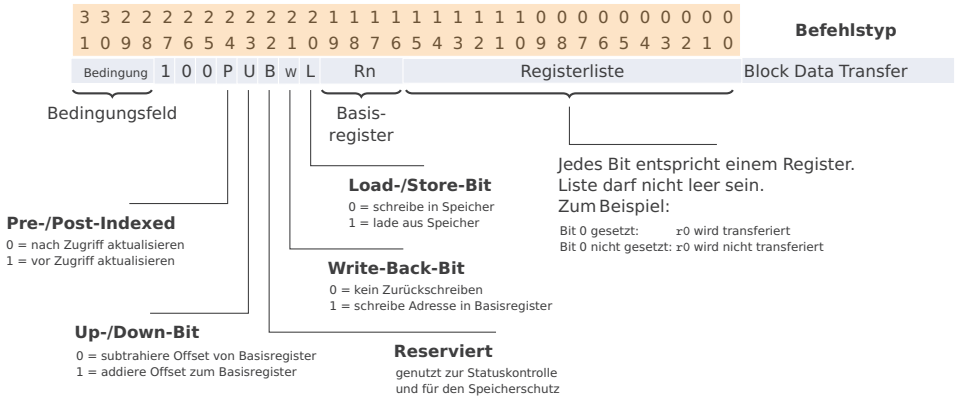
Adressierung erfolgt über Basisregister, jedoch ohne Offset:

```
STM     r0, {r1-r5}     ; r1 bis r5 an die Adressen
                        ; [r0], ..., [r0 + 19] schreiben
LDMIA   r0!, {r3,r6}    ; Register auch einzeln wählbar
```

- Die Reihenfolge ist festgelegt: Speicheradressen steigen mit Registernummer auf.
- Aktualisierung des Basisregisters (Auto-Inkrement) möglich
- Nützlich zum temporären Sichern der Registerinhalte

Beispiel für Kodierung im Instruktionswort

Die Dekodierung erfolgt in der Fetch-Stufe der Prozessor-Pipeline.



Gliederung heute

1. Speicherzugriff
2. **Division und Zahlenausgabe in Assembler**
3. Stapelorganisation und Funktionsaufrufe

Divisionsalgorithmus mit „Restoring“ (W)

Verwendung von bedingter Addition, **Subtraktion** und Schiebeoperationen

Pseudocode

Require: Dividend a, Divisor b (jeweils n Bit)

$(y_{n-1}, \ldots, y_0) \leftarrow a$ {Initialisiere (“load”) $2n$-Bit-Register y.}
$(y_{2n-1}, \ldots, y_n) \leftarrow 0$

for $i = 0$ to $n - 1$ **do**
  $(y_{2n-1}, \ldots, y_0) \leftarrow (y_{2n-2}, \ldots, y_0, 0)$ {Schiebe y um ein Bit nach links.}
  $(y_{2n-1}, \ldots, y_n) \leftarrow (y_{2n-1}, \ldots, y_n) - b$

  if $y_{2n-1} = 0$ **then**
    $y_0 \leftarrow 1$
  else
    $(y_{2n-1}, \ldots, y_n) \leftarrow (y_{2n-1}, \ldots, y_n) + b$ {Wiederherstellung des Rests}
  end if
end for

$r \leftarrow (y_{2n-1}, \ldots, y_n)$
$q \leftarrow (y_{n-1}, \ldots, y_0)$ {Ergebnis. Es gilt: $a = b \times q + r$ }

Realisierung in Assembler

```
                ; Dividend in r1 (16 Bit, vorzeichenlos)
                ; Divisor in r2 (16 Bit, vorzeichenlos)
div:
        MOV     r2, r2, LSL #16
        MOV     r3, #16             ; Schleifenzähler
divloop:
        RSBS    r1, r2, r1, LSL #1  ; schiebe und subtrahiere
        ORRPL   r1, r1, #1
        ADDMI   r1, r1, r2          ; Wiederherstellung des Rests
        SUBS    r3, r3, #1
        BNE     divloop

                ; Quotient in r1_15, ..., r1_0
                ; Rest in r1_31, ..., r1_16

        MOV     pc, lr              ; Rücksprung
```

Ausgabe von Hexadezimalzahlen

Nutzung des Systemaufrufs zur Ausgabe von ASCII-Zeichenketten:

```
                ; Ganzzahl in r4 (32 Bit, vorzeichenlos)
hex:
          MOV   r3, #8              ; 8 Hexadezimalstellen
          MOV   r7, #4              ; wähle Systemaufruf write
          MOV   r2, #1              ; Länge der Zeichenkette
hexloop:
          LDR   r1, =lut            ; Adresse der Zeichentabelle
          ADD   r1, r1, r4, LSR #28 ; addiere Bits 28–31 von r4
          SWI   #0
          MOV   r4, r4, LSL #4      ; nächste Hex-Ziffer in Bits 28–31
          SUBS  r3, r3, #1
          BNE   hexloop

          MOV   pc, lr              ; Rücksprung

lut:      .ascii "0123456789abcdef" ; Look-Up-Tabelle
```

Programmrumpf zum Test von `div` und `hex`

```
.arm                ; assembliere im Standard-ARM-Modus
.text               ; Start eines nicht beschreibbaren Programmbereichs
.global _start                  ; Linker soll Symbol _start kennen
_start:                         ; Konvention für Einsprungpunkt
          LDR     r1, =169      ; Dividend
          LDR     r2, =12       ; Divisor
          BL      div           ; Division: Quotient und Rest in r1

          MOV     r4, r1
          BL      hex           ; Ausgabe

          MOV     r0, #0
          MOV     r7, #1        ; wähle Systemaufruf exit
          SWI     #0

div:      ...       ; von Folie 15

hex:      ...       ; von Folie 16
```

Hörsaalfrage

24 82 94 16

Welche Ausgabe erzeugt das Assemblerprogramm ?

a. `14`

b. `0x000e`

c. `e0001000`

d. `0001000e`

Zugang: `https://arsnova.uibk.ac.at` mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Ausgabe von Dezimalzahlen

```
dec:                ; Ganzzahl in r1 (16 Bit, vorzeichenlos)
          MOV   r8, lr                ; Rücksprungadresse sichern
          LDR   r5, =buffer+5         ; Zeiger auf Ende des Puffers +1
          MOV   r6, #0x30             ; ASCII-Kode für 0 als Offset
          MOV   r7, #0                ; Stellenzähler
decloop:
          ADD   r7, r7, #1            ; nächste Ziffer (mind. eine)
          MOV   r2, #10               ; Basis 10 (dezimal)
          BL    div                   ; r1 : r2 von Folie 15
          ADD   r4, r6, r1, LSR #16   ; Rest als Ziffer in ASCII ...
          STRB  r4, [r5,-r7]          ; ... rückwärts in Puffer schreiben
          BICS  r1, r1, #0x000f0000   ; Rest löschen
          BNE   decloop               ; mehr Stellen wenn Quotient > 0
          SUB   r1, r5, r7            ; Start der Zeichenkette im Puffer
          MOV   r2, r7                ; Länge der Zeichenkette
          MOV   r7, #4                ; Systemaufruf write wählen
          SWI   #0
          MOV   pc, r8                ; Rücksprung
.data               ; für Linker: Start eines beschreibbaren Speicherbereichs
buffer:   .space 5                    ; 5 Byte, denn ⌈log10(2^16)⌉ = 5
```

Gliederung heute

1. Speicherzugriff
2. Division und Zahlenausgabe in Assembler
3. **Stapelorganisation und Funktionsaufrufe**

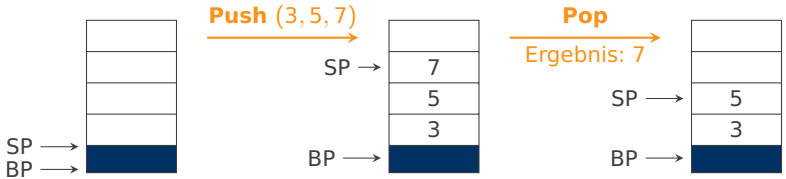
Stapel

Ein **Stapel** (engl. *stack*) ist eine Datenstruktur, die

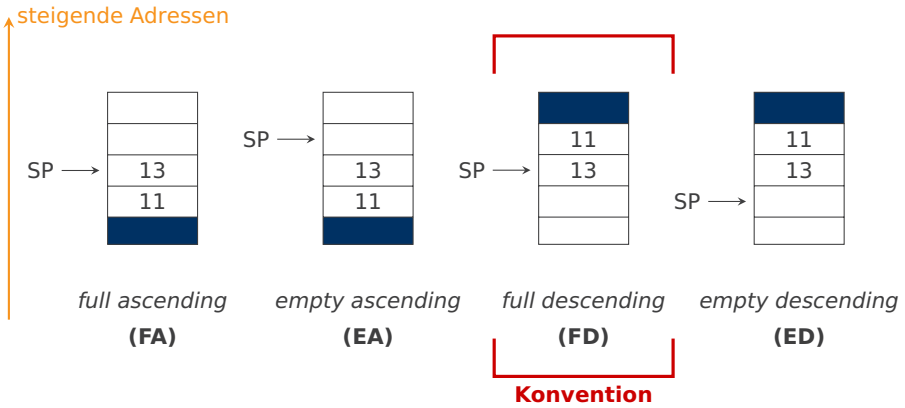
- **wächst**, wenn man neue Daten „darauf“ abgelegt ($\rightarrow$ *push*) und
- **schrumpft**, wenn man Daten „von oben“ wegnimmt ($\rightarrow$ *pop*).

Bei der Realisierung im **Speicher** definieren zwei **Zeiger** (engl. *pointer*) die aktuellen Grenzen des Stapels:

- **Base Pointer** (BP) zeigt auf den „Boden“.
- **Stack Pointer** (SP) zeigt auf die „Spitze“.



Varianten der Stapelorganisation



Realisierung mit dem Block Data Transfer

STM/LDM-Mnemonics können direkt um die Suffixe FA, EA, FD und ED ergänzt werden, um das gewünschte Verhalten zu erreichen.

Nützlich für verschachtelte und rekursive **Unterprogramme:**

```
proc:
        STMFD   sp!, {r0-r12, lr}   ; alle Register
                                    ; einschl. Rücksprungadresse
           ⋮                        ; auf den Stapel legen
        LDMFD   sp!, {r0-r12, pc}   ; wiederherstellen
                                    ; und Rücksprung
```

Beispiel für **Aufruf:**

```
        BL      proc
```

Alternative Suffixe für STM und LDM

Suffix	Bedeutung	verwendet bei
IA	*increment after*	STMEA LDMFD
IB	*increment before*	STMFA LDMED
DA	*decrement after*	STMED LDMFA
DB	*decrement before*	STMFD LDMEA

Anwendung: Skizze einer sehr effizienten Kopierschleife (vgl. Folie 9)

```
blockloop:
            LDMIA   r12!, {r0-r11} ; 48 Bytes laden
            STMIA   r13!, {r0-r11} ; speichern
            SUBS    r14, r14, #1   ; Vielfache von 48
            BNE     blockloop
                  ; vor Rücksprung sp und lr wiederherstellen
```

Allgemeiner Ablauf eines Funktionsaufrufs

1. **Parameter** (Argumente) werden an vereinbarter Stelle (Speicher oder Register) abgelegt
2. Übergabe der Ablaufsteuerung an das Unterprogramm
3. Bereitstellung von Speicher für **lokale Variablen**
4. Vollständige Ausführung der Unterprogramms
5. **Ergebnis** (Wert) wird an Stelle abgelegt, auf welche das aufrufende Programm zugreifen kann
6. Rückgabe der Ablaufsteuerung an das aufrufende Programm; Fortführung an Position unmittelbar nach dem Aufruf

Aufrufkonventionen definieren diese Schnittstelle.

ARM-Aufrufkonventionen

(extrem vereinfacht; Annahme: alle Werte passen in 32 Bit)

Parameter

- Die ersten vier Argumente werden in den Registern `r0`, . . . , `r3` übergeben.
- Alle weiteren kommen auf einen *full descending* Stapel.

Lokale Variablen

- Liegen auf dem Stapel.

Ergebnis

- Rückgabe im Register `r0`.

Das Unterprogramm erhält die Werte aller Register ab `r4`.

Aufruf einer Funktion in der C-Standard-Library

```
.global _start
_start:
        LDR     r0, =msg1 ; 1. Argument (Zeiger auf Zeichenkette)
        BL      printf    ; Aufruf in Bibliothek (→ Linker)

        MOV     r1, r0    ; Rückgabewert als 2. Argument
        LDR     r0, =msg2 ; Format-String als 1. Argument
        BL      printf    ; Ausgabe

        MOV     r0, #0    ; Programm beenden
        MOV     r7, #1
        SWI     #0
                          ; Null-terminierte Zeichenketten
msg1:   .asciz "I love assembler.\n"
msg2:   .asciz "Printed %i characters.\n"
```

Zum Debuggen der C-Schnittstelle:Compiler mit der Option `-S` aufrufen.

Syllabus – Wintersemester 2020/21

07.10.20	1.	Einführung
14.10.20	2.	Kombinatorische Logik I
21.10.20	3.	Kombinatorische Logik II
28.10.20	4.	Sequenzielle Logik I
04.11.20	5.	Sequenzielle Logik II
11.11.20	6.	Arithmetik I
18.11.20	7.	Arithmetik II
25.11.20	8.	Befehlssatzarchitektur (ARM) I
02.12.20	9.	Befehlssatzarchitektur (ARM) II
09.12.20	10.	Prozessorarchitekturen
16.12.20	11.	Ein-/Ausgabe
13.01.21	12.	Speicher
20.01.21	13.	Leistung
27.01.21		**Klausur (1. Termin)**

danach →

inday students virtual edition