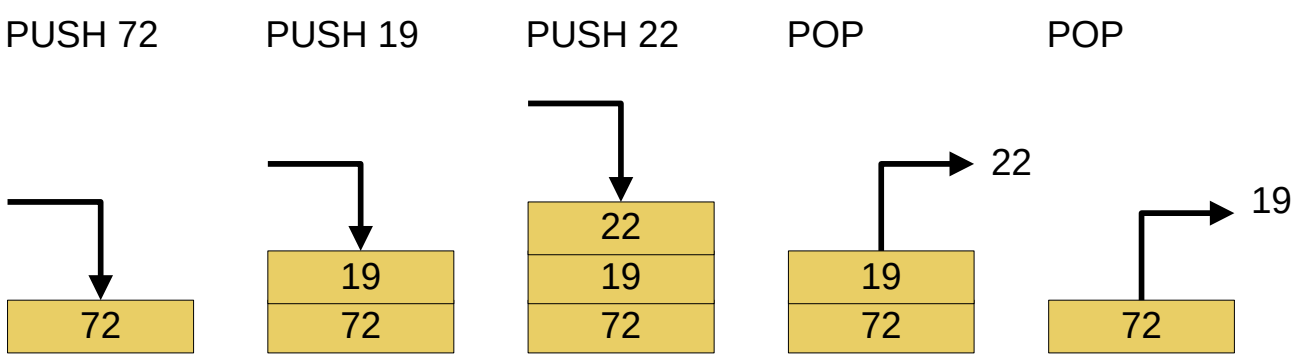
9. Stack und Stackoperationen (Intel IA-32 und Cortex M4)

- Was ist ein Stack?
- Systemstack
- Stackpointer

9.1 Funktionsweise eines Stacks

Stack = Stapelspeicher = Kellerspeicher = LIFO (Last in – First out).

PUSH x	legt ein Element x auf dem Stapel ab
POP	liest (und entfernt) ein Element vom Stapel

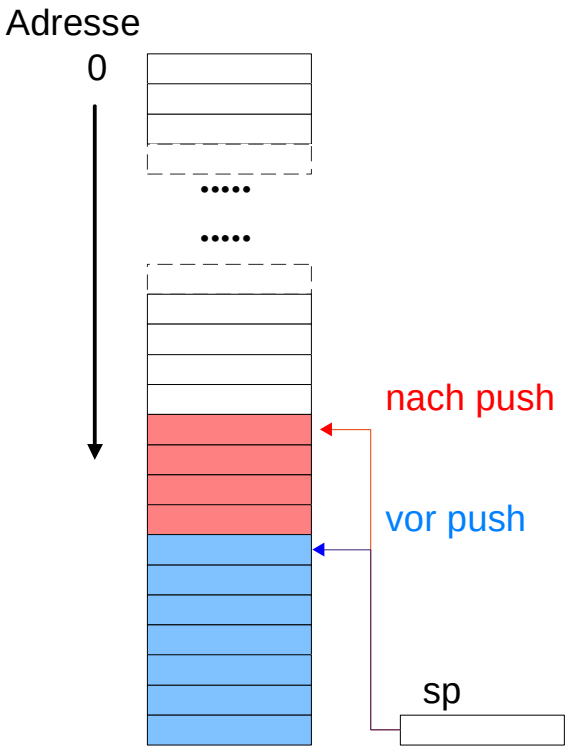


9.2 Systemstack

Das System stellt jedem Programm einen speziellen Speicherbereich zur Verfügung, der als Stack arbeitet. → ***Systemstack***

Dieser Systemstack hat 2 Hauptfunktionen:

- Übergabe von Funktionsparametern,
- Bereitstellen von Speicher für lokale Variablen von Funktionen



Der Stackboden liegt auf einer hohen Adresse und der Stack wächst in Richtung kleinerer Adressen.

Als **Stackpointer** dient das r13-Register (**r13** = **sp**).

Wird also ein Wert auf dem Stack abgelegt (*Push*), dann verkleinert sich der Wert im Stackpointer.

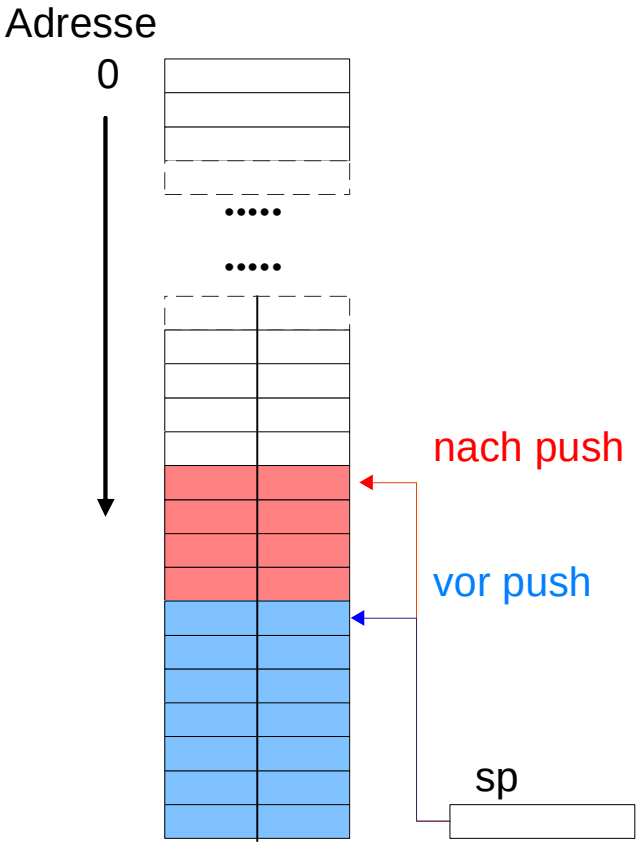
Wird dein Wert vom Stack entfernt (*Pop*), dann vergrößert sich der Wert im Stackpointer.

9.3 Ablegen und Rücklesen von Werten auf dem Systemstack

9.3.1 8-Byte-Alignment des Systemstack (Cortex M4)

PUSH und POP werden wie folgt realisiert:

```
push  {r0, r1, r4-r9}
        . . .
        . . .
pop   {r0, r1, r4-r9}
```



Der Systemstack des Cortex-M4 ist 8-Byte-aligned !!!

Es muss also immer eine gerade Anzahl von Registern (je 4 Byte) auf dem Stack abgelegt werden.

Anm.: ... auch wenn man nur ein Byte retten möchte !

9.3.2 Reihenfolge der Parameter auf dem Stack

push:

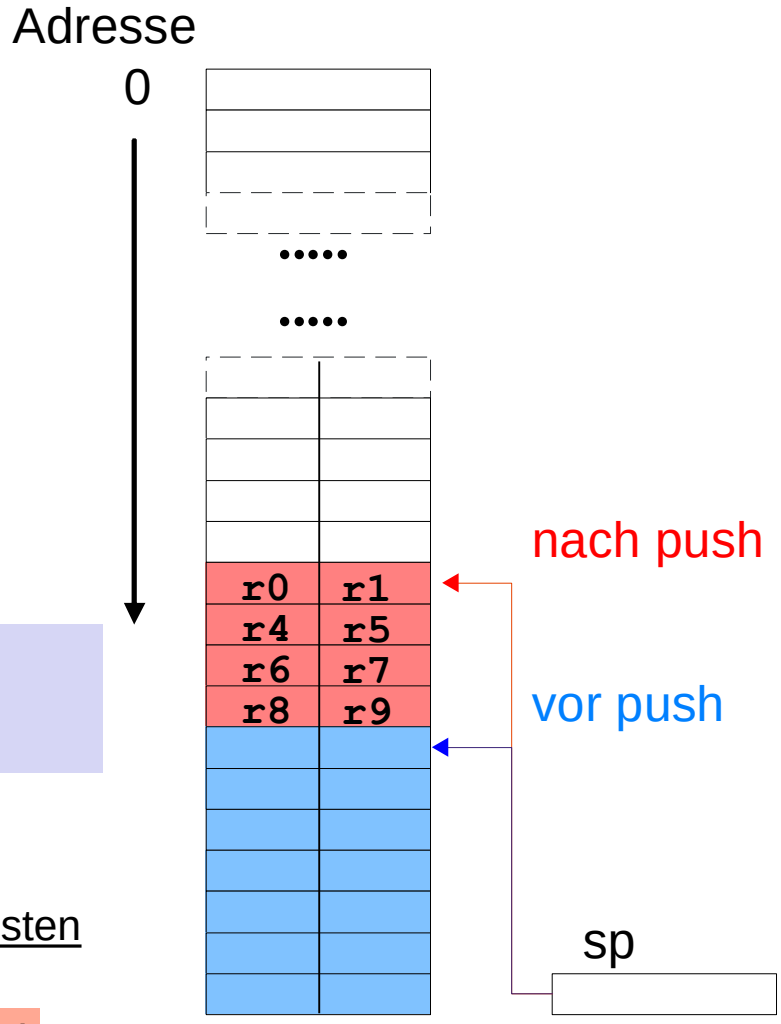
Die Register werden beginnend bei der höchsten Registernummer auf den Stack abgelegt, d.h.

die kleinste Registernummer steht oben auf dem Stack.

Die kleinste Registernummer steht also auf der niedrigsten Adresse.

Beispiel:

```
push  {r1,r0,r9,r8,r4-r7}
```



pop:

Die Register werden beginnend mit der kleinsten Registernummer vom Stack gelesen.

Die Schreibreihenfolge spielt also keine Rolle!

9.3.2 Alternative Befehle für PUSH und POP (z.B. für Userstacks)

PUSH und POP mehrerer Register können auch mit den Befehlen

STMFD (*store multiple, full descending*) und
LDMFD (*load multiple, full descending*)

durchgeführt werden.

```
        stmfd     sp!, {r3-r6, r8, r10-r12}     @ PUSH r12,
                                                @ PUSH r11 .....
        . . .
        . . .
        ldmfd     sp!, {r3-r6, r8, r10-r12}     @ POP r3,
                                                @ POP r4,
```

stmfd: Die Register werden beginnend bei der höchsten Registernummer auf den Stack abgelegt, d.h.

die kleinste Registernummer steht oben auf dem Stack.

10. ARM/Cortex - Prozessor: Unterprogramme

- Zweck von Unterprogrammen
- Übergabemechanismen (globale/lokale Daten, call-by-value, call-by-reference)
- Grundlegende Gedanken
- Thumb-2-Befehle
- Rolle des Stackpointers **sp**
- Registerbehandlung (Übergabe, Retten der Register)
- Parameter- und Ergebnisübergabe

10.1 Fragestellungen zu Unterprogrammen

10.1.1 Wozu Unterprogramme?

Zweck von Unterprogrammen:

- Wiederverwendbarkeit von Codeteilen
- Übersichtlichkeit durch Abstraktion
  - Funktionssammlungen (z.B. Vektorrechnung, trig. Funktionen)
  - Softwareschichten (z.B. Hardwareabstraktionsschichten)
  - Anwendungsmodule (z.B. OCR-Modul, Grafikmodule, ...)

Definition eines Unterprogramms (Pseudocode):

```
Funktionsname (IN: Arg.1, Arg.2    OUT: Erg.1, Erg.2 )
    [Anweisungen]
    ....
RETURN
```

10.1.2 Wie können Funktionsargumente übergeben werden ?

10.1.2.1 Globale Daten

Prinzip: Das aufrufende Programm und das Unterprogramm arbeiten auf den gleichen Daten. Der Speicherort ist hart einprogrammiert und wird nicht explizit übergeben.

Vorteile:

- Der Datenort muss nicht explizit übergeben werden, d.h.
- die Datenübergabe muss nicht implementiert werden.

Nachteile:

- Es ist nicht offensichtlich, mit welchen Daten das Unterprogramm arbeitet.
- Schlecht wiederverwendbar, da das Unterprogramm das Vorhandensein der globalen Daten voraussetzt (*feste Kopplung*).
- Gefahr von Seiteneffekten, da man schnell die Übersicht darüber verliert, wer die Daten verändert und wann sie verändert werden.

10.1.2.2 Call-by-value

Prinzip: Das aufrufende Programm übergibt dem Unterprogramm die notwendigen Eingangsdaten als Kopie.

Vorteile:

- Ohne Seiteneffekte, da das Unterprogramm keinen Zugriff auf externe Daten hat.
- Der anwendende Programmierer benötigt kein Internwissen über das verwendete Unterprogramm (Geheimnisprinzip = information hiding).

Nachteile:

- Auf einzelne Werte beschränkt.
  Strukturierte Datentypen (z.B. Strings, Vektoren, Arrays, ...) können nur sehr umständlich und ineffizient übergeben werden.

10.1.2.3 Call-by-Reference

Prinzip: Das aufrufende Programm übergibt dem Unterprogramm die Adresse der Ein-/Ausgangsdaten.

Vorteile:

- Kontrollierbare Seiteneffekte, da der Programmierer bei Anwendung des Unterprogramms explizit den Zugriff auf die externen Daten erlaubt.
- Der anwendende Programmierer benötigt kein Internwissen über das Unterprogramm (Geheimnisprinzip = information hiding).
- Auch strukturierte Datentypen sind leicht übergebbar.

Nachteile:

- Restgefahr von Seiteneffekten bleibt, da das Unterprogramm externe Daten verändert.

10.1.3 Wie müsste ein Unterprogrammaufruf arbeiten ?

10.1.3.1 Vorüberlegung

Sprung zum Unterprogramm:

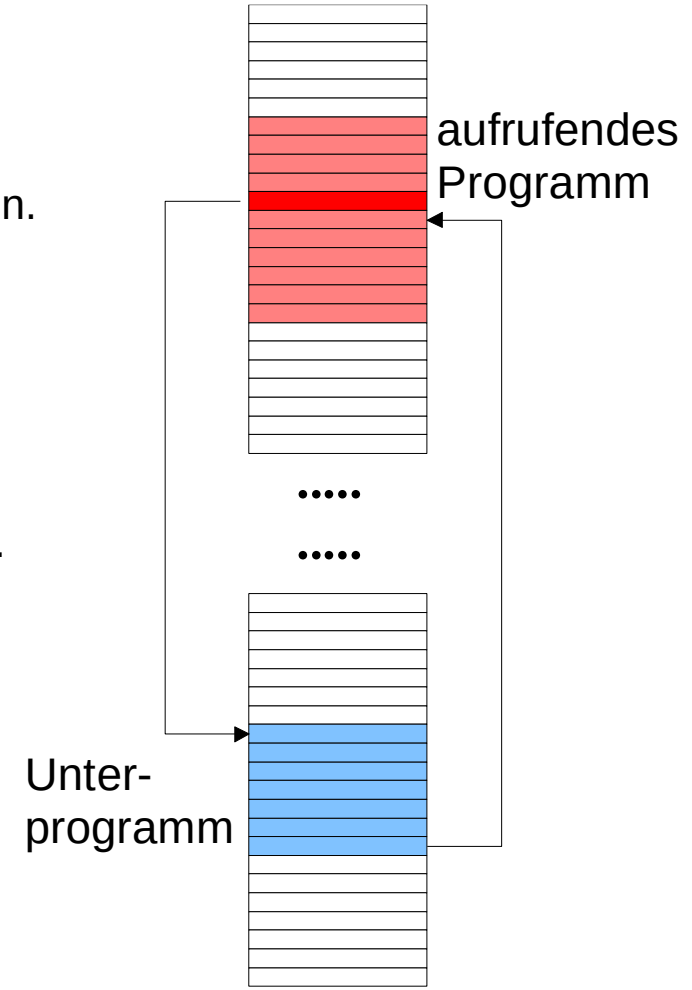
Um zu einem Unterprogramm zu verzweigen, muss der *Program Counter* (**pc** = **r15**) auf den ersten Befehl des Unterprogramms gesetzt werden.

Rücksprung vom Unterprogramm:

Um mit dem Kontrollfluss nach Abarbeitung des Unterprogramms wieder an der Aufrufstelle fortzusetzen, muss der Program Counter auf den dem Aufruf nachfolgenden Befehl gesetzt werden.

Fazit:

Mit dem Sprung in ein Unterprogramm muss die Rückkehradresse zum aufrufenden Programmteil gespeichert werden.



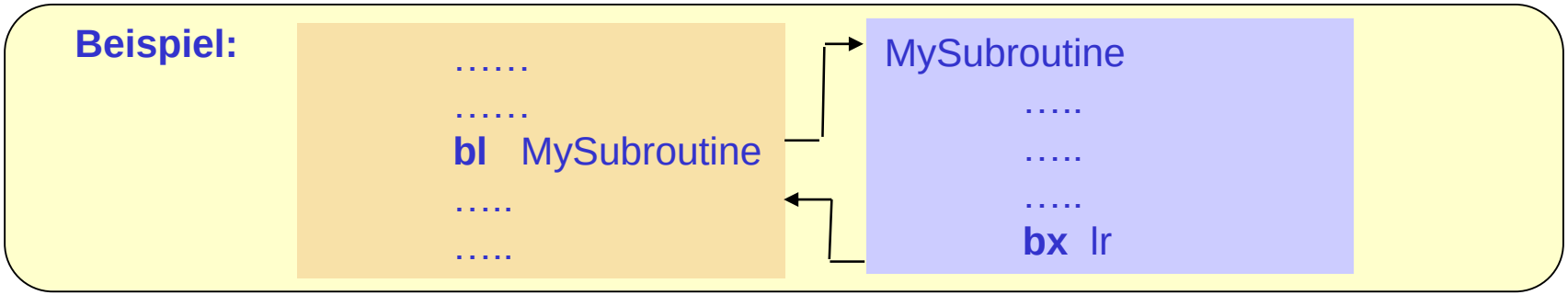
10.1.3.2 Befehlssequenz

Um das Problem des Speicherns der Rücksprungstelle zu lösen, verwendet man die folgenden Befehle:

- **bl** <Label> : Transfer des Kontrollflusses zum Unterprogramm (*relative branch with link*)
- **bx** lr : Rückkehr zum Hauptprogramm (*branch and exchange*)

bl rettet die Programmadresse nach dem Sprungbefehl (pc) im **Linkregister** (**lr** = **r14**) Danach wird die Programmausführung an der Unterprogrammadresse fortgesetzt!

„bx lr“ kopiert die in lr gespeicherte Rücksprungadresse wieder in den pc.



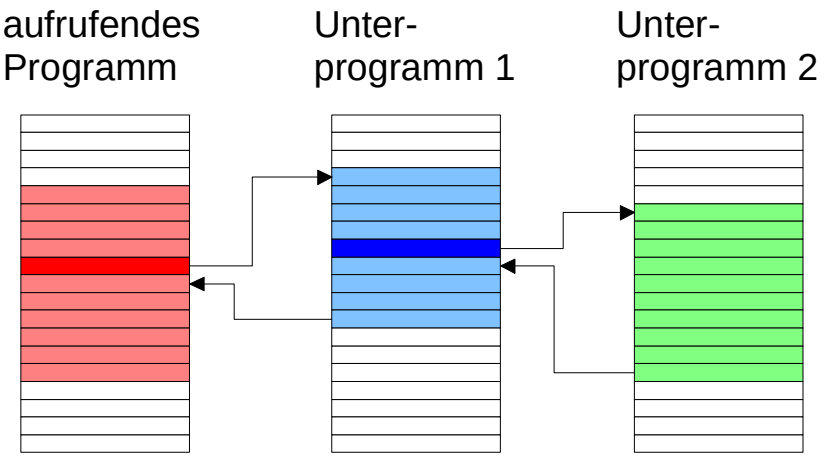
Nur z. Info: `bl` erlaubt nur Sprünge von +/-32MB. „*Long branches*“ sind möglich durch:

```
mov lr, pc
ldr pc, =MySubroutine
```

10.1.4 Geschachtelte Unterprogrammaufrufe (nested calls)

10.1.4.1 Vorüberlegung

Das Prinzip der Sicherung und Rekonstruktion von Rücksprungadressen muss auch im Fall beliebig geschachtelter Unterprogrammaufrufe funktionieren.



Was ist hier das Problem?

Vorschlag zur Lösung?

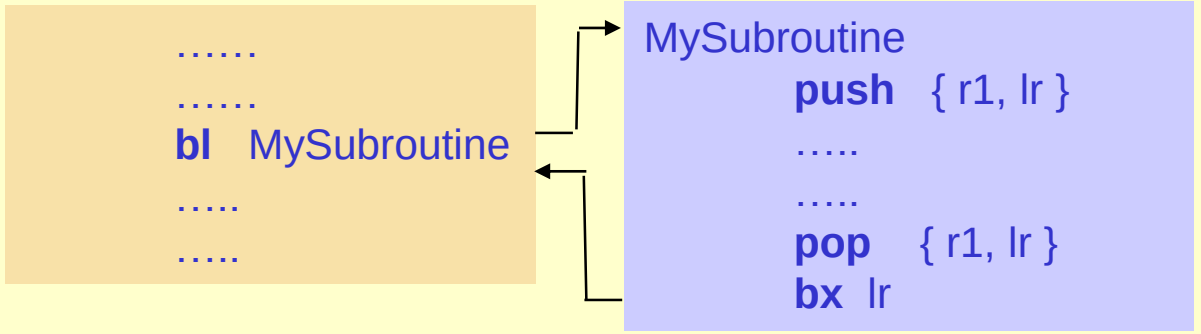
Sicherung und Rekonstruktion haben LIFO-Charakter (Last-in, First-out).
Ein Stack ist also die ideale Datenstruktur für diese Aufgabe.

10.1.4.2 Befehlssequenz

Im Unterprogramm

- wird zunächst das Linkregister auf den Stack gerettet (push),
- der Unterprogrammcode wird ausgeführt und anschließend
- wird das Linkregister wieder restauriert (vom Stack gelesen, pop).

Beispiel:



Achtung: Es ist zu bedenken, dass immer eine gerade Anzahl von Registern auf den Stack gerettet wird ! (deswegen hier : push {r1, lr})

Alles ok? Noch ein Problem?

10.1.5 Registerrettung

10.1.5.1 Vorüberlegung

- Beim Unterprogrammsprung wird nichts gerettet:

  - keine Registerinhalte,
  - keine Condition codes,
  - keine lokale Variablen.

Wie rettet man Register ?

Also: Der Programmierer trägt die Verantwortung für die Rettung von Registerinhalten, Condition codes und lokalen Variablen. !

- Stack !

- Aber: Der Stackpointer **sp** selbst sollte nur für Unterprogramme genutzt werden. Anderenfalls ist die Gefahr falscher Rücksprungadressen sehr groß.

Also: Finger weg vom Register sp (Stackpointer). !

10.1.5.2 Retten der verwendeten Register

Konsistenz der CPU-Register wahren

Zur Vermeidung unerwarteter Effekte (Seiteneffekte) nach Unterprogrammen sollte darauf geachtet werden, dass die im Unterprogramm verwendeten Register nach dem Rücksprung aus dem Unterprogramm wieder die gleichen Inhalte haben wie vor dem Unterprogrammaufruf.

Lösung: Die verwendeten Register zu Beginn des Unterprogramms auf den Stack retten und vor dem Rücksprung zum Hauptprogramm wieder restaurieren.

Ausnahme: Die für die Parameterübergabe verwendeten Register.

Beispiel:

```
……
……
bl   MySubroutine
…..
…..
```

```
MySubroutine
        push  {r1-r3, lr}
        …..
        …..
        pop   {r1-r3, lr}
        bx  lr
```

10.2 Weitere konzeptionelle Gedanken zu Unterprogrammen

10.2.1 Verschiedenen Übergabemechanismen

Der Rückgabewert einer Funktion wird meist über Register **r0** zurückgegeben.
Für die Eingabewerte (Parameter) stehen folgende Mechanismen zur Verfügung:

Was \ *Wie*	**Register** r0, r1, r2, r3 (ARM) - begrenzte Parameterzahl	**Stack** + kaum begrenzte Parameterzahl
Datenkopie *Call-by-value* + Seiteneffektfrei + information hiding - nur einfache Datentypen	CbV-Reg	CbV-Stk
Adressen *Call-by-reference* + beliebige Datentypen + information hiding - Restgefahr von Seiteneffekten	CbR-Reg	CbR-Stk

ÜBUNG: Parameterübergabe: „Call-by-value“ über Register (CbV-Reg)

Schreiben Sie ein Unterprogramm, welches die *Fakultät* n! zu einer gegebenen Zahl n berechnet.

Verwenden Sie das Register r0 zur Übergabe des Eingabeparameters (n) und des Ergebnisses (n!).

Was sind die Beschränkungen dieser Methode?

ÜBUNG: Parameterübergabe: „Call-by-reference“ über Register (CbR-Reg)

Schreiben Sie ein Unterprogramm, welches die Länge eines Strings bestimmt.

Zur Übergabe des Strings soll das Register r0 verwendet werden.
Das Ergebnis soll ebenfalls in r0 stehen.

Was sind die Beschränkungen dieser Methode?

10.2.2 Parameterübergabe über den Stack

10.2.2.1 Einleitende Anmerkungen

Eine übliche Konvention beim ARM Thumb-Befehlssatz ist, die ersten vier Parameter über r0 – r3 zu übergeben und alle weiteren Parameter über den Stack.

Dies ermöglicht für die meisten Unterprogramme (0 ... 4 Parameter) einen sehr schnellen Unterprogrammsprung (mit wenig Overhead), ermöglicht aber auch mehr (5 ... beliebig viele) Übergabeparameter (mit etwas mehr Overhead).

Bei vielen anderen Prozessoren (mit weniger Registern) ist die Parameterübergabe über den Stack der Standard.

Die Parameter r4 - r11 stehen üblicherweise als lokale Variablen zur Verfügung.

Werden mehr lok. Variablen benötigt, so kann auch hierfür der Stack verwendet werden (s.u.).

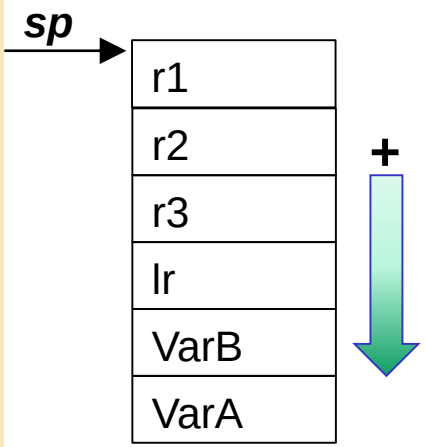
Die Parameterübergabe nach Konvention ist dann einzuhalten, wenn die Assembler-Unterprogramme auch von C-Programmen aufgerufen werden sollen.

10.2.2.2 Realisierungsidee 1 (nicht gut)

```
; Aufrufendes Programm

    push   {r3, r4}          ; PUSH VarA und VarB (r4, r3)
    bl     MySubroutine
    add    sp, #8            ; Stack korrigieren
```

MySubroutine:
- 2 Parameter
- Übergabe über Stack
- intern genutzte Register: r1 und r2
- Ergebnis in r0

```
; geändertes Unterprogramm

MySubroutine
    push    {r1-r3, lr}      ; PUSH, Register retten
    ldr   r2, [sp, #16]      ; [r2] ← VarB
    ldr   r1, [sp, #20]      ; [r1] ← VarA
    …….                      ; irgendwas berechnen ……
    …….                      ; und Ergebnis → [r0]

    pop   {r1-r3, lr}        ; POP,  Register restaurieren
    bx    lr
```



Worin liegt der Nachteil dieses Ansatzes ?

Werden im Unterprogramm weitere Register auf den Stack kopiert, dann müssen alle Versatzwerte (*Offsets*) korrigiert werden !!

→ schwer wartbar
→ sehr fehleranfällig

10.2.2.2 Realisierungsidee 2 (besser, aber auch noch nicht gut) → fp

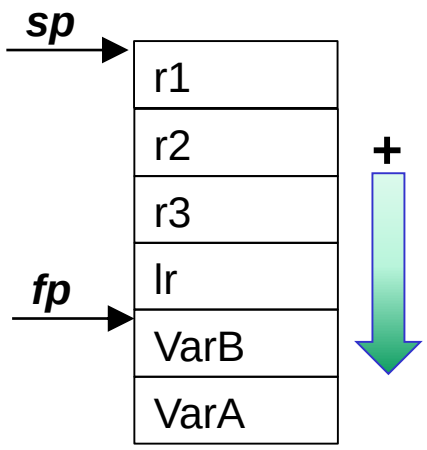
```
# Aufrufendes Programm
    push   {r3, r4}          ; PUSH VarA und VarB (r4, r3)
    bl     MySubroutine
    add    sp, #8            ; Stack korrigieren
```

MySubroutine:
- 2 Parameter
- Übergabe über Stack
- intern genutzte Register: r1 und r2
- Ergebnis in r0

```
Unterprogramm

MySubroutine
    mov     fp , sp          ; aktuelle Stackpos. merken
    push    {r1-r3, lr}      ; PUSH, Register retten
    ldr     r2, [fp, #0]     ; [r2] ← VarB
    ldr     r1, [fp, #4]     ; [r1] ← VarA
    .......                  ; irgendwas berechnen ……
    .......                  ; und Ergebnis → [r0]

    pop   {r1-r3, lr}        ; POP,  Register restaurieren
    bx      lr
```

fp ist der sog. ***Framepointer***

fp = r11



Worin liegt der Nachteil dieses Ansatzes ?

Im Falle von „*nested calls*“ wir der alte Inhalt des Framepointer fp überschrieben.

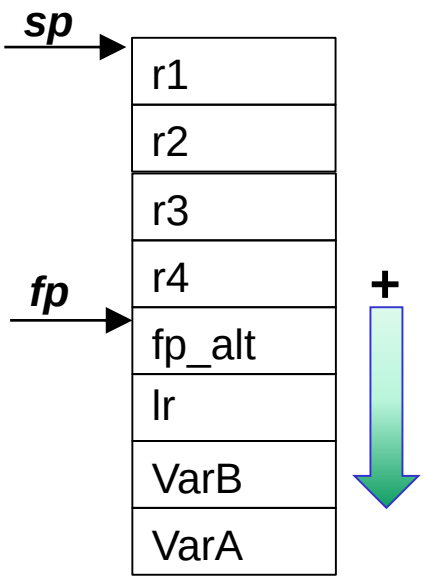
→ nicht schachtelbar

*10.2.2.3 Realisierungsidee 3 (**so geht's !!!**)*

```
# Aufrufendes Programm

    push   {r3, r4}          ; PUSH VarA und VarB (r4, r3)
    bl     MySubroutine
    add    sp, #8            ; Stack korrigieren
```

MySubroutine:
- 2 Parameter
- Übergabe über Stack
- intern genutzte Register:
  r1- r4
- Ergebnis in r0

```
Unterprogramm

MySubroutine

    push   {fp, lr}          ; PUSH fp , fp_alt retten
    mov    fp , sp           ; aktuelle Stackpos. merken
    push   {r1-r4}           ; PUSH,  Register retten
    ldr    r2, [fp, #8]      ; [r2] ← VarB
    ldr    r1, [fp, #12]     ; [r1] ← VarA
    .......                  ; irgendwas berechnen ……
    .......                  ; und Ergebnis → [r0]

    pop    {r1-r4}           ; POP,  Register restaurieren
    pop    {fp,lr}           ; POP  fp u. lr restaurieren
    bx     lr
```



ÜBUNG: Parameterübergabe über Stack (1)

Schreiben Sie ein Unterprogramm, welches folgende Berechnung durchführt:

Erg = (z1+z2)*(z3+z4)

- Die Parameter z1 … z4 sollen über den Stack übergeben werden.
- Das Ergebnis soll über r0 zurückgeben werden.
- Alle verwendeten Register sollen gerettet und wieder restauriert werden.