Kapitel 2: Assemblerebene

Instruction Set Architecture (ISA)

...ist die gemeinsame Mikroarchitektur einer Prozessorfamilie, charakterisiert durch:

- Instruktionen welche gibt es?
- Instruktionsformat wie sind sie codiert? # der Operanden
- Adressierungsarten immediate, indirekt ...
- Interrupt-Logik (Unterbrechungsbehandlung)
- Speichermodell (z.B. linear, segmentiert, seitenbasiertes)

Befehlsformat

Wird auch durch die maximalen Anzahl von Operanden einer Instruktion klassifiziert:

- 3-Adress-Befehle ($\Rightarrow$ 3-Adress-CPU/-Rechner)
  - `ADD R0,R1,R2` $\Rightarrow$ `R0 := R1 + R2`
  - meist Load-Store-Architekturen:

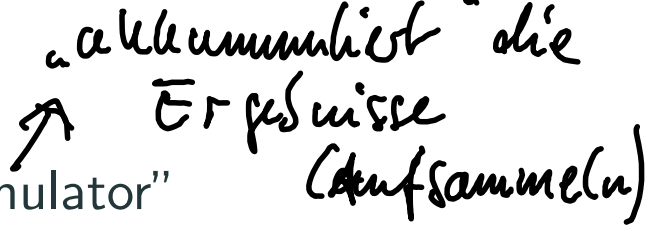
Register füllen $\rightarrow$ nur auf Registeren rechnen $\rightarrow$ Register zurückschreiben

(Hauptspeicher-Zugriffe *nur* über Load-/Store-Instruktionen)

Befehlsformat (2)

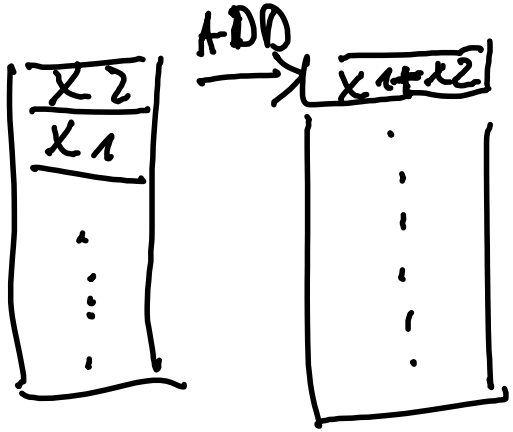
- 2-Adress-Befehle
  - Ein Operand ist ggf. implizites Ziel
  - `ADD R0,R1` $\Rightarrow$ `R0 := R0 + R1`

- 1-Adress-Befehle
  - Ein Operand vollständig implizit: oft "Accumulator"
  - `ADD R1` $\Rightarrow$ `R0 := R0 + R1` bzw. `Acc := Acc + R1`



Befehlsformat (3)

- 0-Adress-Befehle
  - Stack-basierte CPU
  - ADD $\Rightarrow$ `(TOS) := (TOS) + (TOS-1)`
    TOS = Top of Stack



Bsp. Java VM!
(mit Teilen (Arithmetik + Ergebnisse)
weitere Hochsprache, Stack-basiert:
FORTH

Befehlsklassen

Data Movement Register $\leftrightarrow$ Register , Register $\leftrightarrow$ Hauptspeicher,...

HS

⇒ in LOAD-/STORE-Arch: einzige Zugriff auf HS

Arithmetical/Logical Rechnen, logische Verknüpfungen

⇒ ALU (Integer, Float oft sep. (s.u.))

Shift / Rotate Oft ebenfalls ALU, ggf. spezielle Shift-Unit

⇒ meist auch ALU, teilw. separate Shift-Units

Control Transfer Sprünge, Verzweigungen, Subroutine Calls (bedingt / unbedingt)

↳ Möglichkeit, nicht-linear Ausführung

Befehlsklassen (2)

DSP-Style **D**igital **S**ignal **P**rocessor

Floating Point Bei μControllern z.T. in Coprocessor-Einheiten

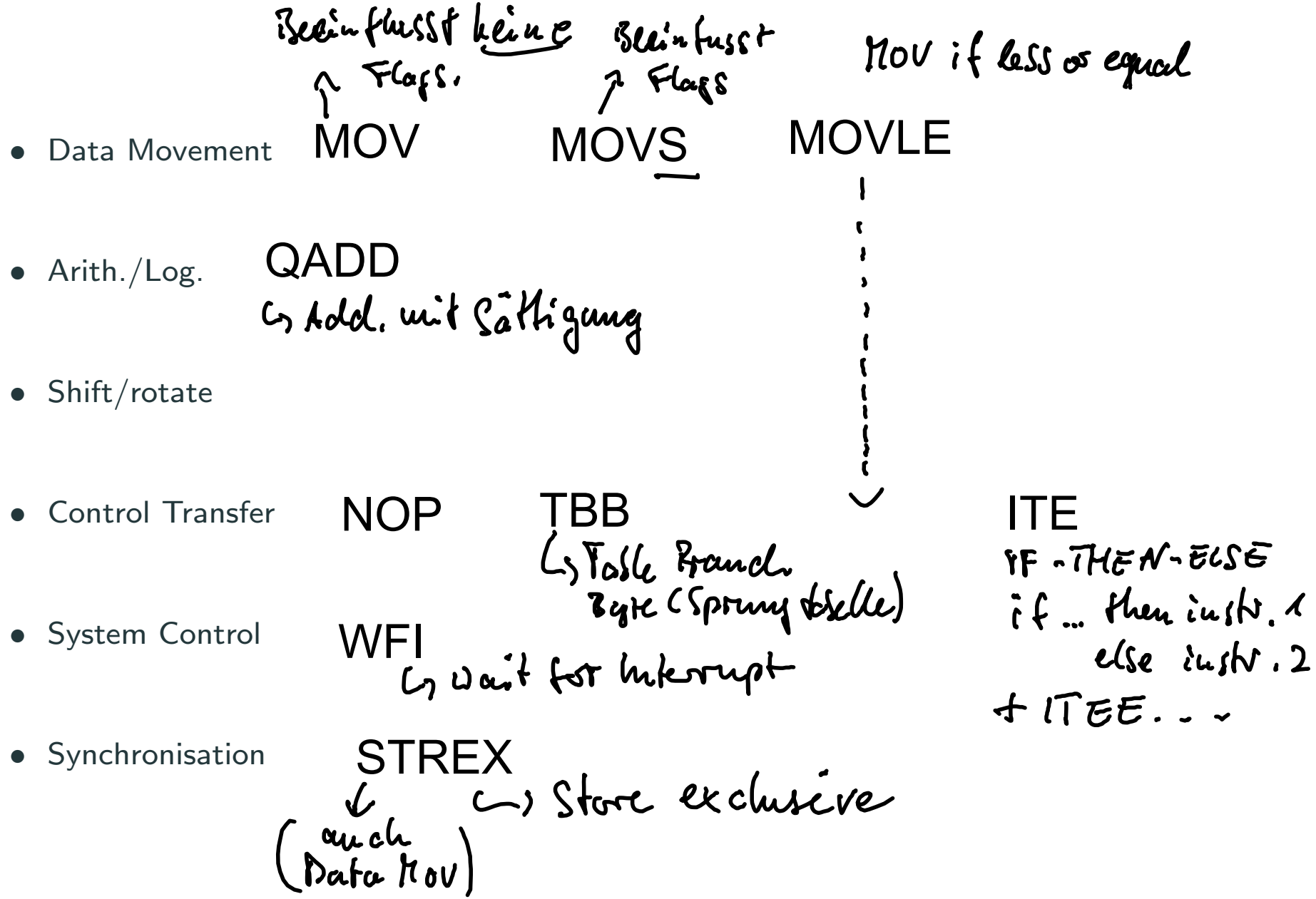
System Control Sleep Modes, Reset, WDT bedienen

Synchronisation atomarer Speicherzugriff, Realisierung und Schutz kritischer Abschnitte

Beispiel: AVR

- Data Movement MOV LD ST BLD CBR
- Arith./Log. ADD EOR INC CPI ASR
- Shift/rotate LSR
- Control Transfer RCALL BRGE RET
- System Control SLEEP BREAK NOP
- Synchronisation XCH

Beispiel: ARM



Adressierungsarten

- **Register**
  $R_1 \rightarrow R_2$
- **Immediate** (Konstante)
  $const \rightarrow R_1$

- **Absolute** (Konstante)
  $Mem(Addr) \rightarrow R_1$

Adressierungsarten (2)

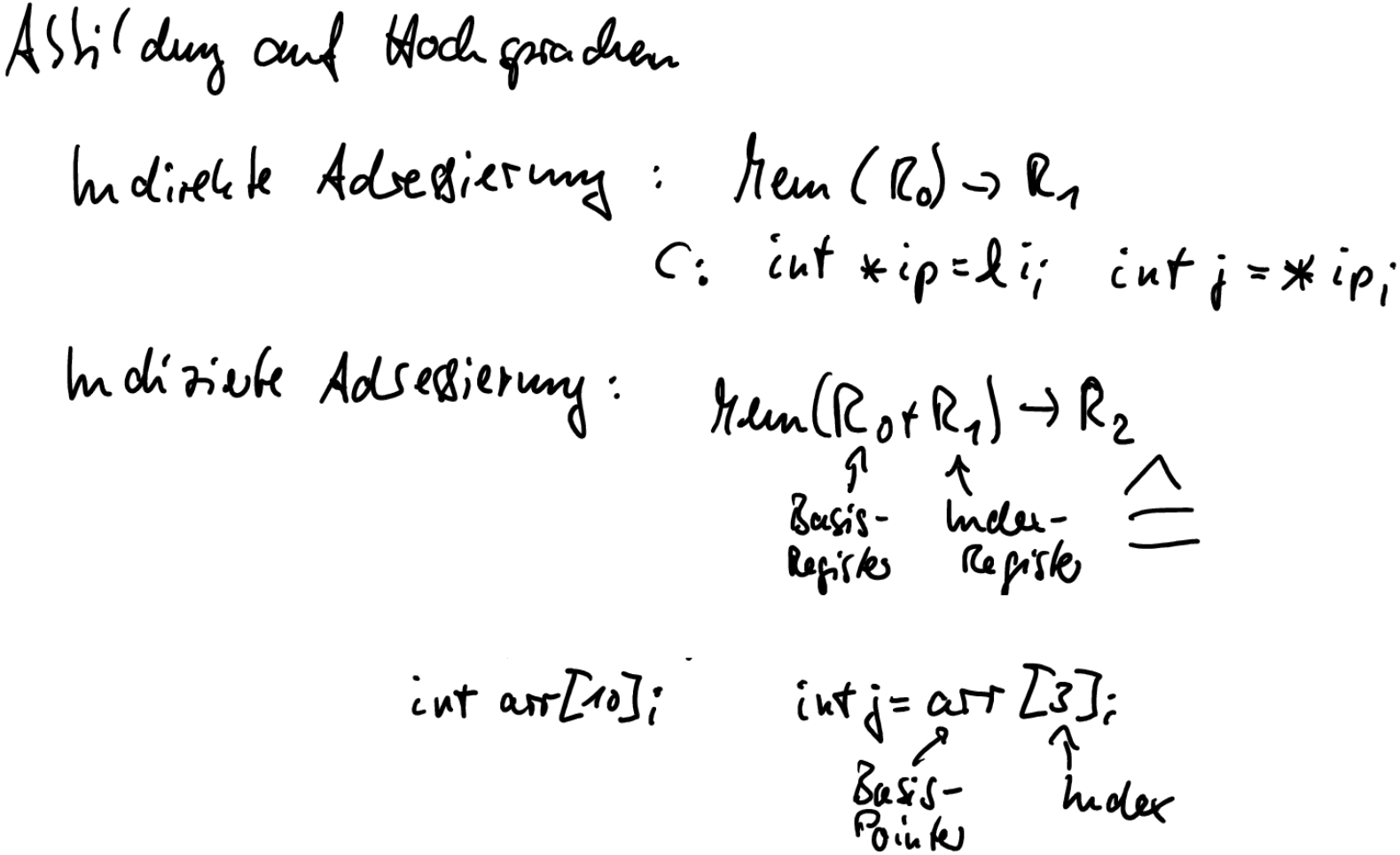
- **Indirect**
  $Mem(R_0) \rightarrow R_1$

  Varianten:
  - mit +/- **Offset**-Konstante
    $Mem(R_0 + const) \rightarrow R_1$
  - **pre/post In-/Dekrement**
    $Mem(R_0 + const) \rightarrow R_1$
    $R_0 \; += \; postInkrConst$

Adressierungsarten (3)

- **Indexed** (Indiziert)
  $Mem(R_0 + R_1 + const) \rightarrow R_2$
- **PC-Relative**
  $Mem(PC + offset) \rightarrow R_0$
- **PC Indirect**
  $R_0 \rightarrow PC$

Adressierungsarten (4)



Conditionals

Bedingungen für die Ausführung einer Instruktion, z.B. BEQ
Branch if equal

	Bedingung	Flags	
EQ	equal	(Arg1 - Arg2 = 0)	(Vergleiche basieren auf Subtraktion)
NE	not equal		
MI	minus		
PL	plus		
VS	overflow set		
VC	" cleared		
HI	higher	(unsigned)	
LS	lower or same	(unsigned)	
GE	greater or equal	(signed)	entweder Rechnung korrekt, $t2 \geq t1$, oder Rechnung falsch => N false
LT	less than	(signed)	
GT	greater	(signed)	
LE	less or equal	(signed)	
(AL)	always		(Flags wie ARM)

Unterprogrammaufrufe

Abgrenzung von einfachen Sprüngen:

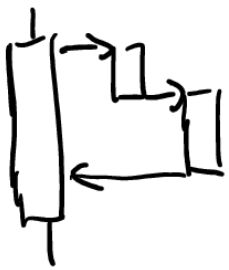
Subroutine Call with return

→ Sprung in Subroutine

← Rücksprung

- Schachtelung (strikt) über mehrere Ebenen möglich
- Rücksprungadresse muss gesichert werden

Simple jump/branch



- kein Rücksprung eingeplant

Subroutinen-Instruktionen AVR vs. ARM

AVR: (r/c)call : Aufruf, legt Adresse nach call auf dem Stack
ret : Lädt Adresse vom Stack → PC (pop)

Schachtelung implizit unterstützt

ARM: BL (branch link): Aufruf, lädt LR mit Adresse nach BL

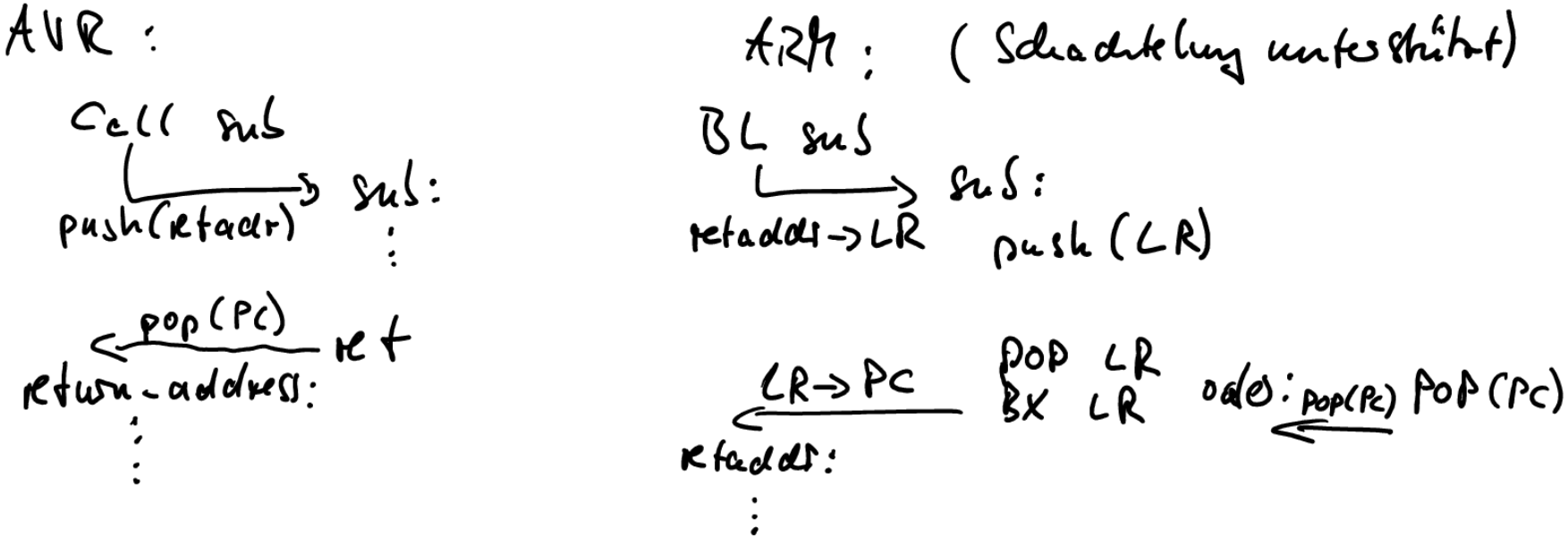
Link Register, R14

BX LR : LR → PC

Schachtelung erfordert Sichern von LR (auf Stack):

Problem: keine Schachtelung möglich

Subroutinen-Aufruf: AVR/ARM



Parameterübergabe

- Problem: Übergabe Input-Parameter → Funktion
  Rückgabewert ←
- Bsp. C: Compiler generiert Code für Caller + Callee
  ⇒ Übergabe einheitlich, funktioniert, aber - Compiler-abhängig
  - Schlecht reproduzierbar
- Konventionen nötig, sobald Module separat compiliert und später gelinkt werden
- 3 Möglichkeiten
  - globale Variablen (schlecht)
  - Stack
  - Register

  } oft gemischt: Register effizienter, aber begrenzt
  ⇒ Stack hinzunehmen wenn Register nicht ausreichen

Varianten der Parameterübergabe via Stack

Probleme bei Stackübergabe

Reihenfolge?

Wer räumt auf (Stackaufbau)?

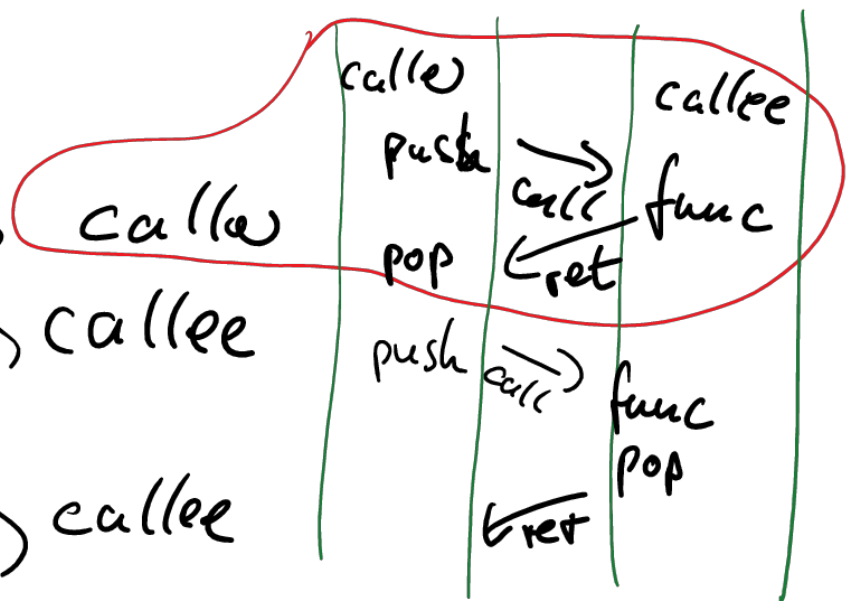
rechts-nach-links (a,b) ⇒ push b, push a

cdecl - - - - - - > callee

stdcall - - - - - - > callee

links-nach-rechts (a,b) ⇒ push a, push b

pascal - - - - - - > callee



AVR-libc-Konventionen

Eigtl ∃ kein "Atmel"-Standard, de facto wurde die Konvention von gcc/libc übernommen

R0 : temporary

R1 : 0 als Konstante

call-used : kann von Subroutine verändert werden: R18-27, R30/31

call-saved : Subroutine muss Wert sichern (muss nach Aufruf unverändert sein) R2-17, R28/29

AAPCS

ARM Standard for Application Procedure Calls

Argumentregister "A1-A4" ⇒ R0 - R3 } können von Funkt. verändert werden
Rückgaberegister R0 + R1

Variable Reg. "V1-V5" ⇒ R4 - R8 } müssen von Funkt gesichert werden
V7/V8 ⇒ R10/R11