Mikroprozessorsysteme I

Vorlesung:

Dozent: **Dr. Horsch**

Sprechstunde: siehe Homepage
t.horsch@fbi.fh-darmstadt.de, D14 Raum 1.09, Tel: 16 8449
www.fbi.fh-darmstadt.de/~horsch/mps1

Organisation

Prüfung: **Termin legt Prüfungsausschuss ca. 6 Wochen vor Semesterende (siehe Aushang)**

Leistungsnachweis:

- schriftliche Prüfung (zugübergreifend)
- Zulassung: Fachgespräch im Praktikum (6 von 6 Testaten)

Praktikumsziel:

- Umgang mit dem ARM Befehlssatz
- Umgang mit Entwicklungswerkzeugen

Praktikumsdurchführung:

- jeweils 16 Personen (8 Gruppen zu je 2 Personen)
- 7 Termine
- Vorbereitung notwendig
  - vertraut machen mit Aufgabenstellung
  - Lösungsansätze aufzeigen zu Beginn des Praktikums
  - stichprobenhafte Prüfung
  - bei wiederholtem “nicht vorbereitet sein” ist Ausschluss möglich
- Tausch nur am 1. Termin mit Partner möglich
- Anwesenheitspflicht (Ausnahme: Krankheit / Attest)

Vorlesungsinhalt

Themen:
- Einführung, einfacher Prozessor
- Arithmetik, Logik
- Load / Store, Stackkonzepte,....
- Speicherkonzepte, Datenaufteilung,Initialisierung, Labels
- Einfache Programmstrukturen (if then else, while, switch,...)
- Unterprogramme (Branch & Link, APCS)
- Umsetzung von C-Routinen in ARM Assembler
- Prozessordesign, RISC / CISC
- Ausblick auf Mikroprozessortechnik II

Literatur:
- Steve Furber, ARM Rechnerarchitekturen für System-on-Chip Design (FH-Bibliothek)

ARM Development Kit (auf meiner Seite)
Knoppix CD mit ARM Entwicklungswerkzeugen (Hr. Pester, Fachschaft)

Praktikumsinhalt

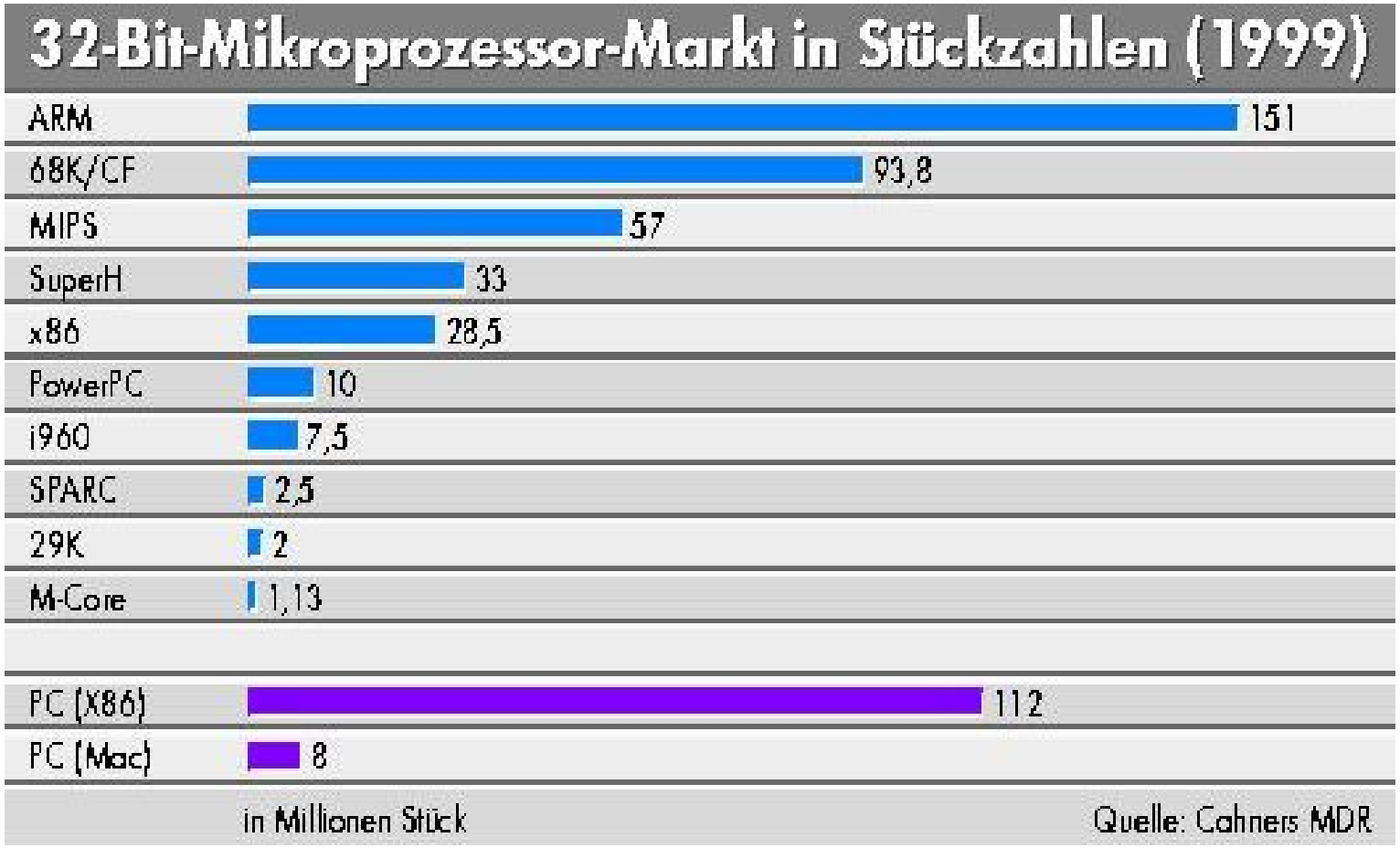
- Termin 1: Einführung in Editor, Compiler, Tools (kein Testat)

- Termin 2: Befehlssatz I (Arithmetik, Logik)
- Termin 3: Befehlssatz II (Load/Store, Branch, Conditional Befehle,...)
- Termin 4: Konzepte: Speicherkonzept, Datenaufteilung, Initialisierung
- Termin 5: Einfache Programmstrukturen (switch, if then else, while,...)
- Termin 6: Einfache Algorithmen u. Datenstrukturen (z.B. Sortieren)
- Termin 7: Unterprogramme (APCS, Branch & Link, rekursive Funktionen)

Mikroprozessormarkt

Praktikum: ARM7TDMI wird benutzt

Bedeutung von ARM (Advanced Risc Machines)



Ein einfacher Prozessor

Grundlegende Komponenten:

Programmzähler-Register (PC)

Speicherung der Adresse des aktuellen Befehls

Register Akkumulator (ACC)

enthält zu bearbeitenden Datenwert

Ein Rechenwerk (ALU)

Operationen auf (binäre) Operanden: SUB, ADD, …

Befehlsregister (IR)

enthält den aktuellen Befehl während dessen Ausführung

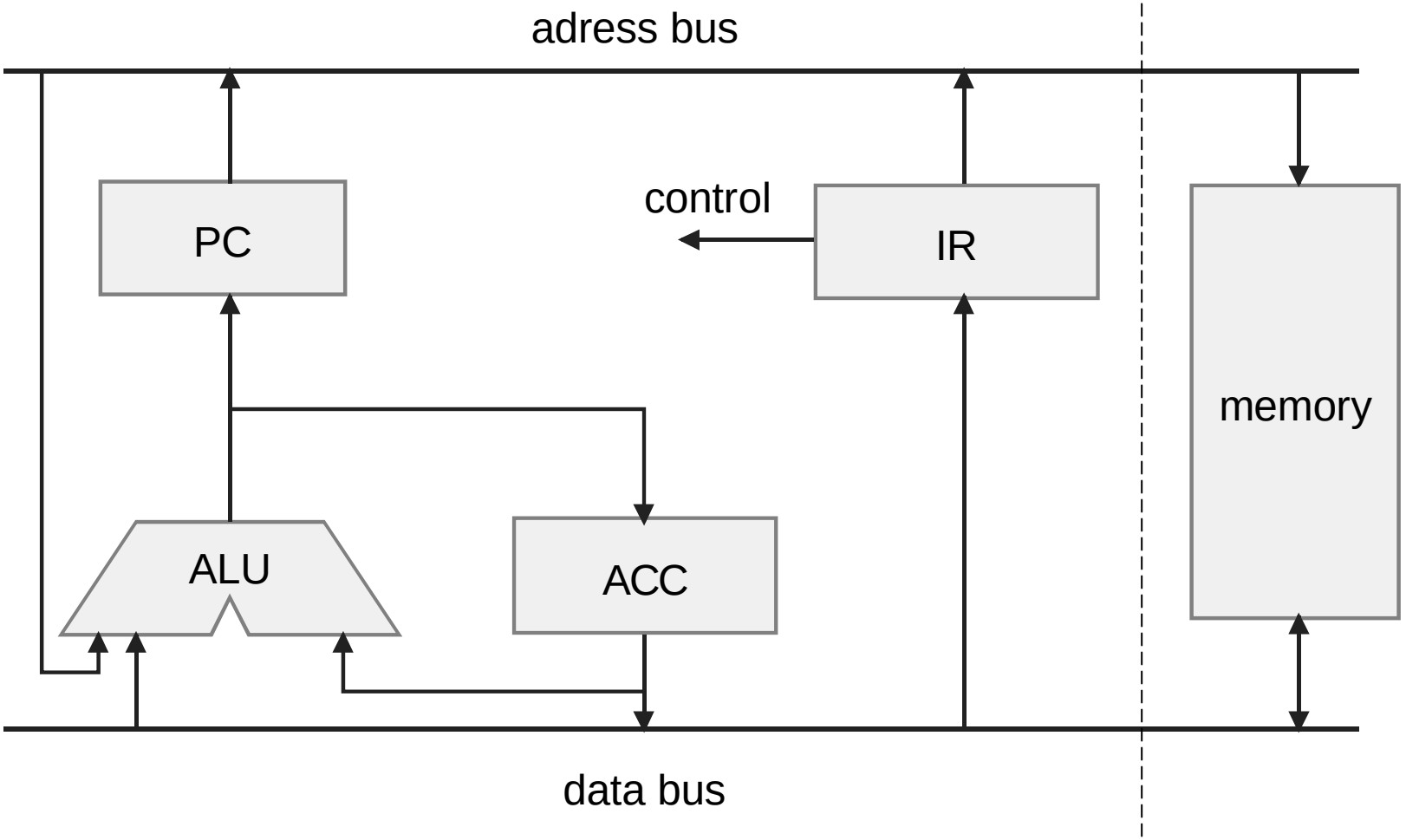
Implementierung eines eingeschränkten Befehlssatzes

16-Bit-Prozessor mit 12 Bit Adressraum

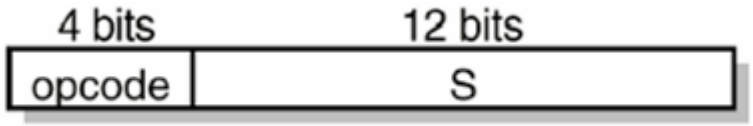
Speicher: 4096 einzeln adressierbaren 16-Bit-Speicherstellen

=> 8kByte adressierbarer Speicher

Ein einfacher Prozessor (MU0)



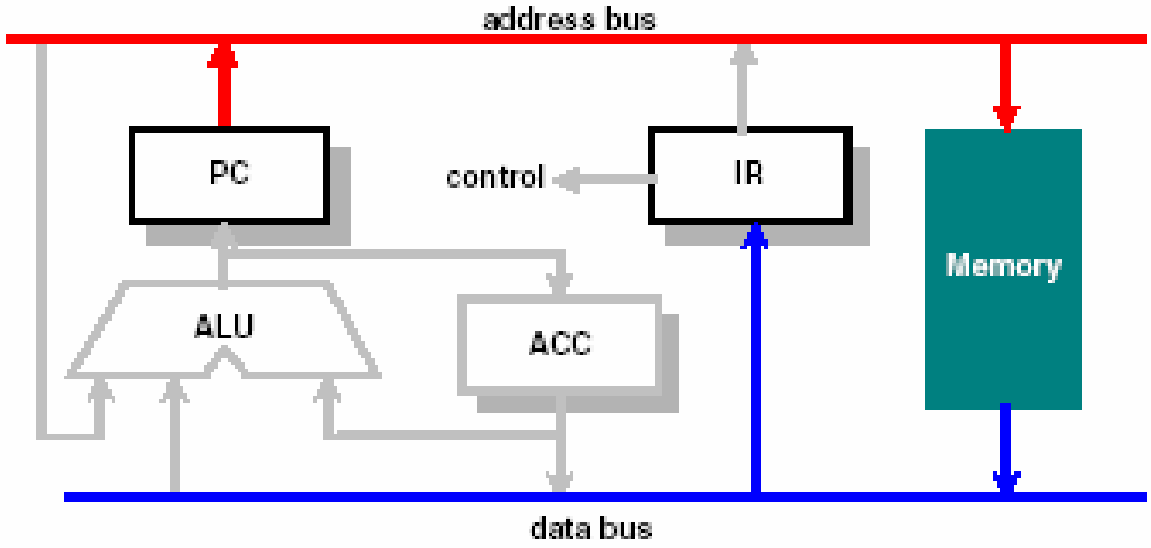
MU0 Befehlssatz



Befehlsformat

Instruktion	Op-Code	Effekt
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC - mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	Falls ACC $\geq$ 0 PC := S
JNE S	0110	Falls ACC $\neq$ 0 PC := S
STP	0111	Stop

Befehl holen



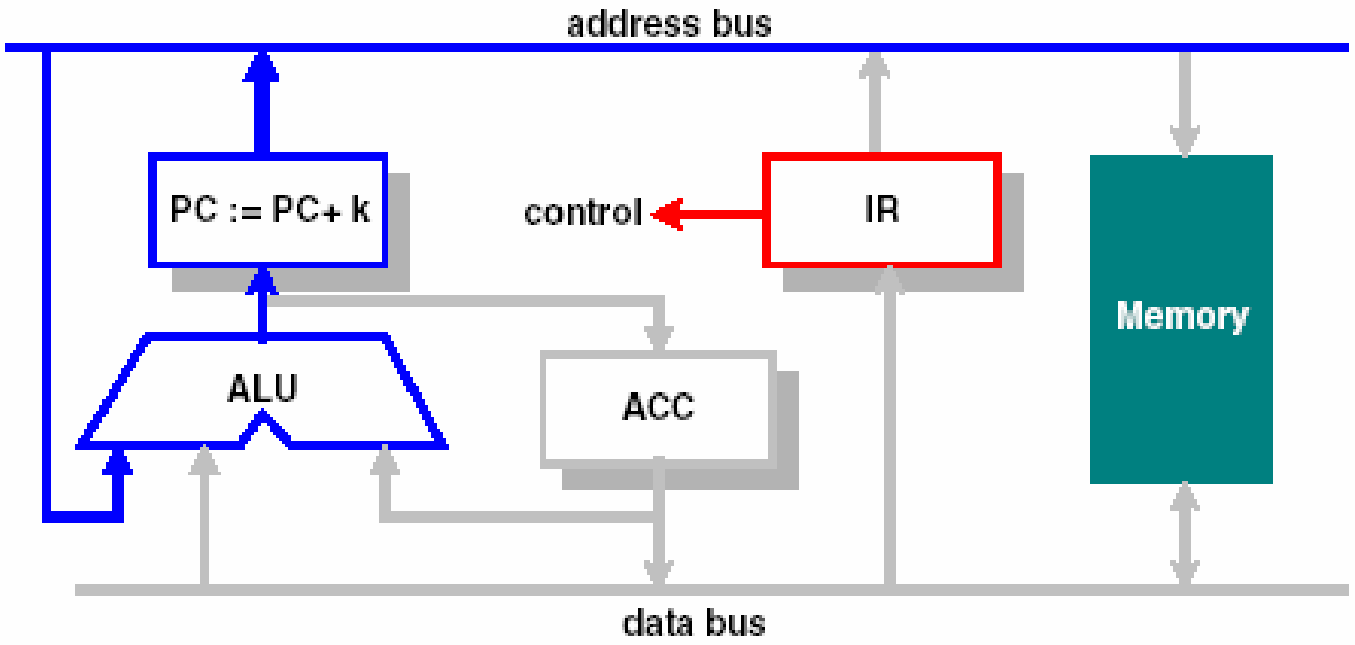
1. Schritt: Hole nächsten Befehl

Prozessor gibt Befehlzähler (PC) auf den Adressbus aus

Speicher legt den Inhalt an der Befehlsadresse auf dem Datenbus

Befehl wird im Befehlsregister gespeichert

Befehl dekodieren

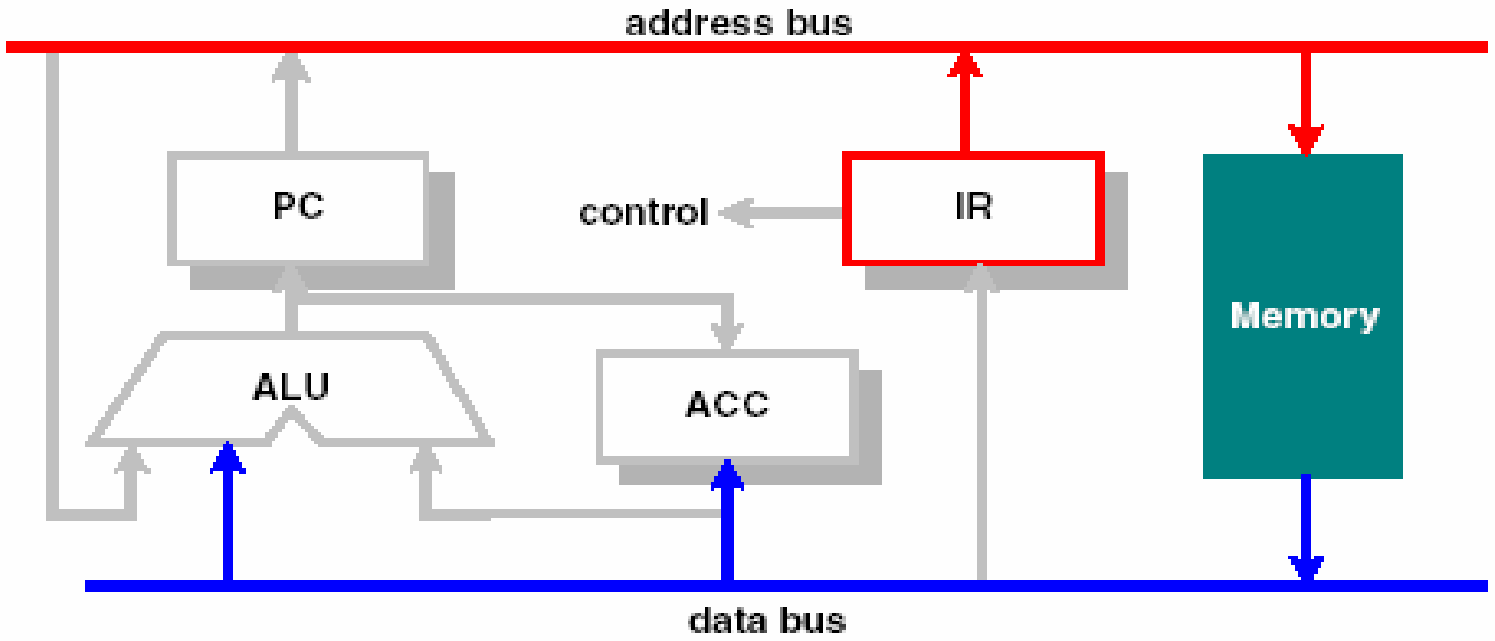


2. Schritt: Befehl dekodieren

Befehl im Befehlregister wird dekodiert vom Schaltwerk

Befehlszähler wird von der ALU inkrementiert

Operand holen

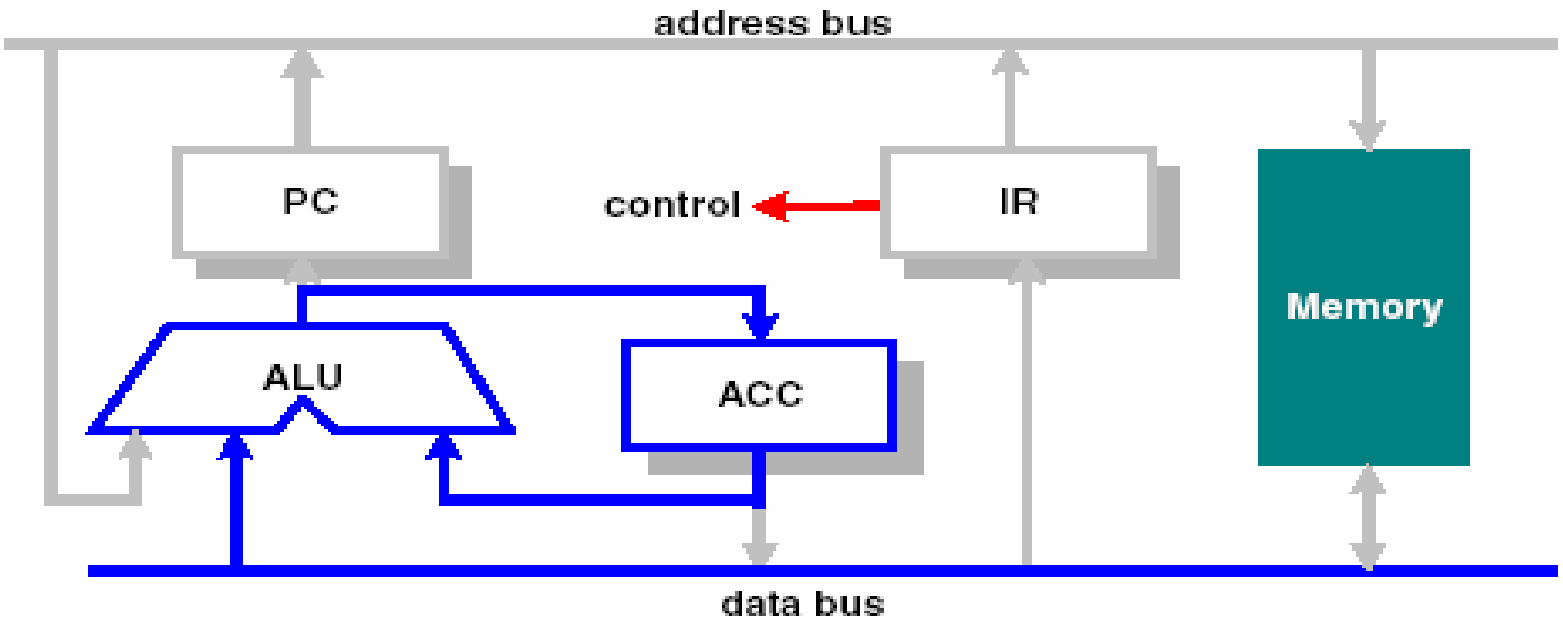


3. Schritt: Operand holen

Adresse vom Operand im Befehlregister wird auf den Adressbus gelegt

Speicher legt den Inhalt an der Operandenadresse auf dem Datenbus

Befehl ausführen

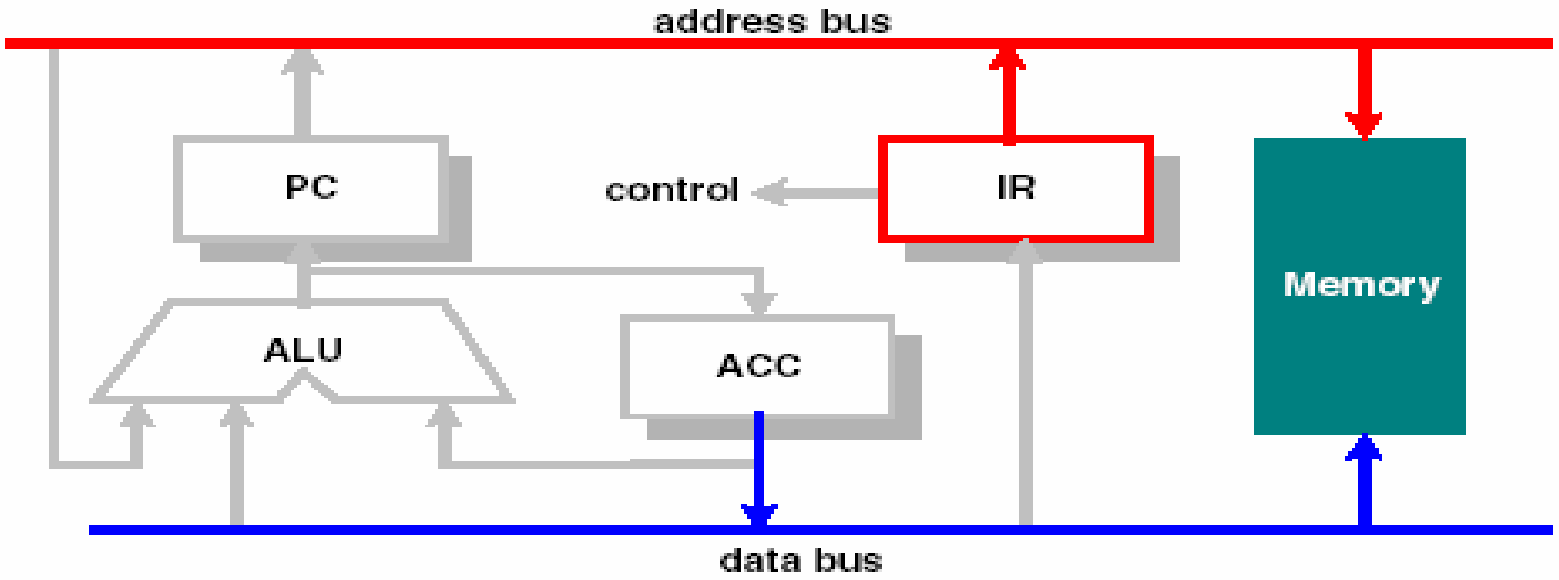


4. Schritt: Befehl ausführen

Befehl wird von der ALU ausgeführt

Result wird im Akkumulator (ACC) gespeichert

Ergebnis speichern



Ergebnis speichern (STO S)

Adresse vom Operand im Befehlregister wird auf den Adressbus gelegt

Wert im Akkumulator wird über den Datenbus in den Speicher gelegt

Zusammenfassung

Befehl hat typischerweise 2 Komponenten

- OpCode (bestimmt **was** getan werden soll)
- Operand (bestimmt **wo** die Daten liegen)

Befehlszähler (Register PC)

- zeigt auf den aktuellen Befehl im Speicher
- wird automatisch inkrementiert

Annahme:

Anzahl der Takte des Befehls ist gleich Anzahl der notwendigen Speicherzugriffe

LDA, STO, ADD, SUB	2 Takte
JMP, JGE, JNE, STP	1 Takte

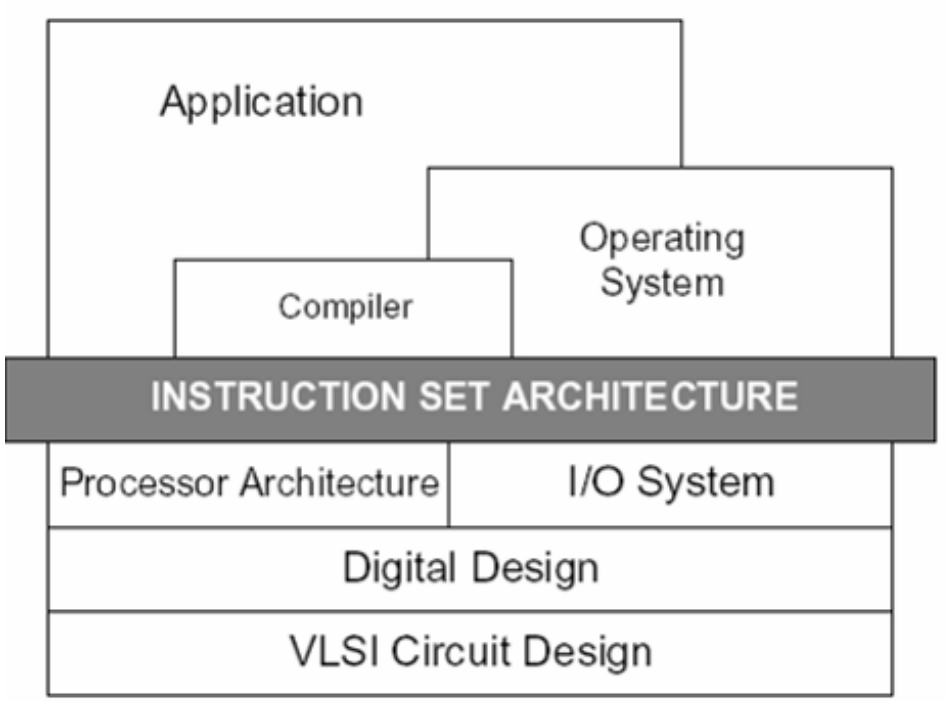
Beispiel: f = (a+b) – (c+d);

MU0 Design Entscheidungen

- Datenpfad Operationen
  - Jeder Befehl wird in 2 Schritten ausgeführt
    - 1. Schritt: Benutze den Memory Operanden und führe die Aktion aus
      - Die Adresse im Instruktionsregister wird benutzt um entweder ein Datum aus dem Speicher zu holen und die Operation auszuführen oder um den Akkumulator zu speichern
    - 2. Schritt: Hole die nächste Instruktion die ausgeführt werden soll
      - Entweder wird der aktuelle PC Inhalt geladen, inkrementiert und wieder zurückgeschrieben, oder die nächste PC Adresse wird aus dem Instruktionsregister geholt inkrementiert und wieder zurückgeschrieben
- Initialisierung
  - Der Prozessor muß in einem bekannten Zustand starten
    - Nach einem Reset startet der Prozessor von der Adresse 0

Bedeutung des Befehlssatzes (ISA)

- Befehlssatz = Instruction Set Architektur (ISA)
- Der ISA Level ist das Interface zwischen der Software und der Hardware
- Der ISA Level definiert die Sprache, die sowohl von der Software als auch von der Hardware verstanden werden muß.



Gute Instruction Set Architekturen

Eine gute ISA sollte …

- einen Satz von Befehlen definieren, der effizient mit heutigen und zukünftigen Technologien zu implementieren ist, so daß kostengünstige Designs über mehrere Generationen enstehen können.
- ein gutes Ziel für compilierten Code sein.
- sowohl die Hardwaredesigner glücklich machen (einfach zu implementieren) als auch die Softwaredesigner (einfach dafür Code zu generieren)

Die Wirklichkeit

- Zur ISA dürfen bei jedem neuen Prozessor Befehle hinzugefügt werden, Hauptsache die Kompatibilität zur alten Software bleibt erhalten

Eigenschaften des ISA Levels

- **Der ISA Level ist definiert durch das Bild wie sich der Prozessor dem Programmierer darstellt.**
- **Der ISA Level Code ist der Code, den ein Compiler generiert.**
- **Bestandteile des ISA Levels**
  - Speichermodell
  - Register
  - Datentypen
  - Befehle
  - ...

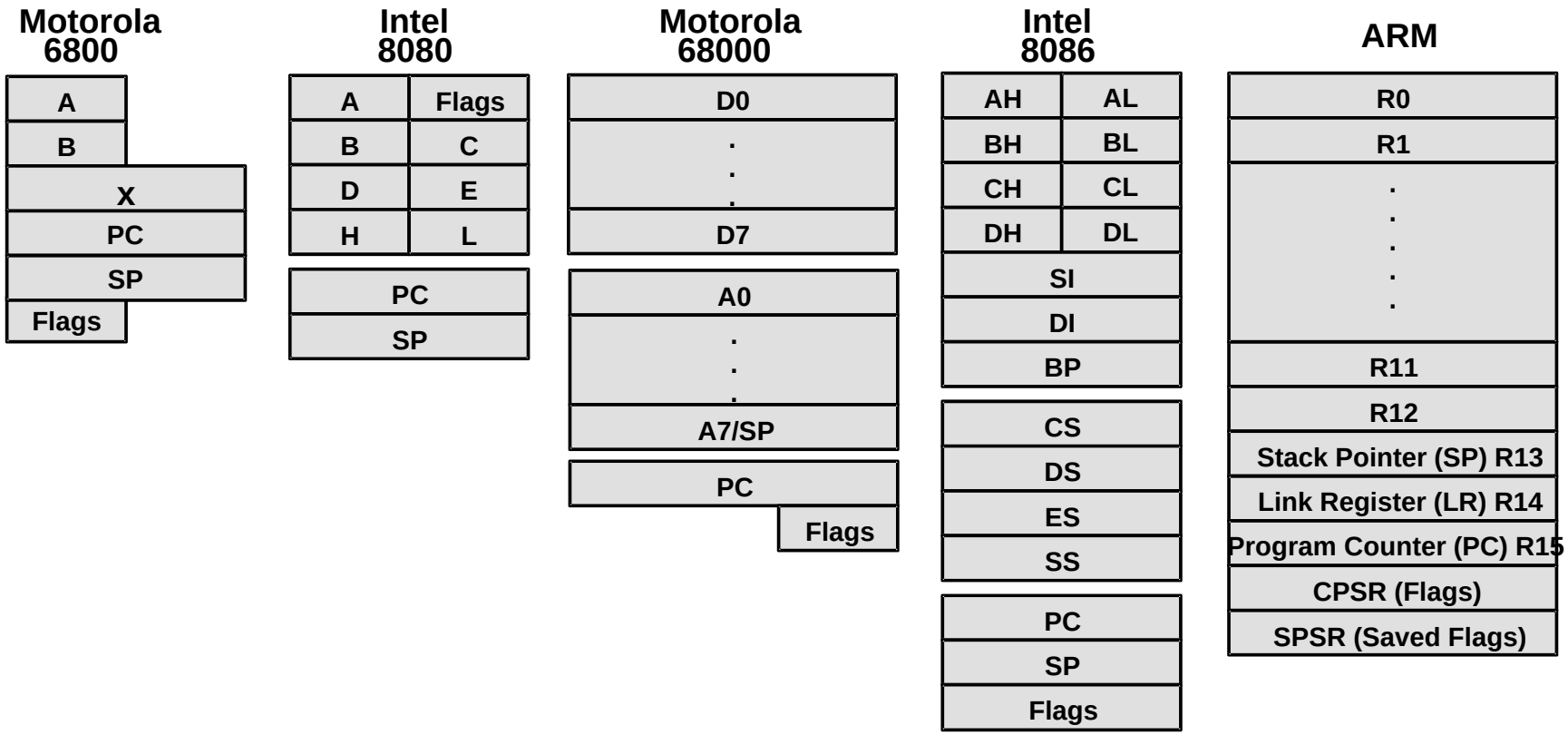
Register

- **Special Purpose Register**
  - Program Counter
  - Stack Pointer
  - Program Status Register
  - ...
- **Universalregister**
  - für die Zwischenspeicherung von Variablen
  - und Zeiger

Das Prozessorstatuswort

- Die Condition Codes eines Prozessorstatusworts
  - N — Gesetzt, wenn das Resultat Negativ war
  - Z — Gesetzt, wenn das Resultat Null war
  - V — Gesetzt, wenn das Resultat einen oVerflow hervorrief
  - C — Gesetzt, wenn das Resultat einen Übertrag aus dem letzten Bit erzeugte
  - A — Gesetzt, wenn ein Übertrag aus dem Bit 3 auftrat
  - P — Gesetzt, wenn das Resultat gerade Parität hatte

Registermodelle im Vergleich



wichtige Register

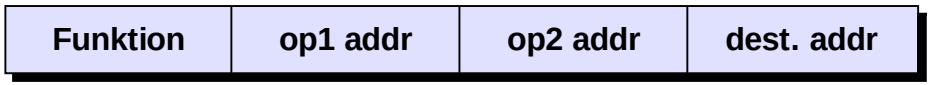
- Akkumulator
  - Universalregister für arithmetische und logische Operationen
- Datenregister
  - Speicherung von Daten
  - meist sind alle Akkumulatorfunktionen möglich
- Adressregister
  - Spezialregister für Index und Adressoperationen
- PC Program Counter
  - enthält die Adresse der nächsten zu ladenden Anweisung.
  - Wird bei Sprunganweisungen und Unterprogrammaufrufen geändert

- SP Stack Pointer
  - zeigt auf den Bereich für temporäre Variable
- Basepointer, Framepointer
  - Zeiger auf den lokalen Datenbereich einer Funktion. Kann direkt an den Stackpointer gekoppelt sein.
- Control Register
  - hier werden Konfigurationen für den Prozessor eingestellt
- Status Register
  - enthält binäre Informationen über die letzten Operationen

Befehlssatz Formate

4 Adress Instruktionsformat (Microcode)

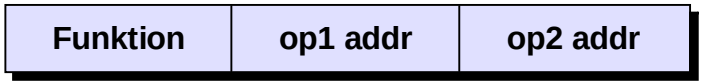
Add d, s1, s2, next_i; d = s1 + s2



3 Adress Instruktionsformat (Standard ARM, SPARC, MIPS)

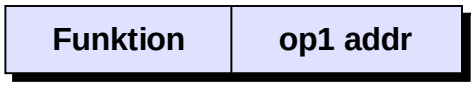
Add d, s1, s2; d = s1 + s2

Instruktion Set Formate



2 Adress Instruktionsformat (Intel, Motorola 68K, ARM Thumb)

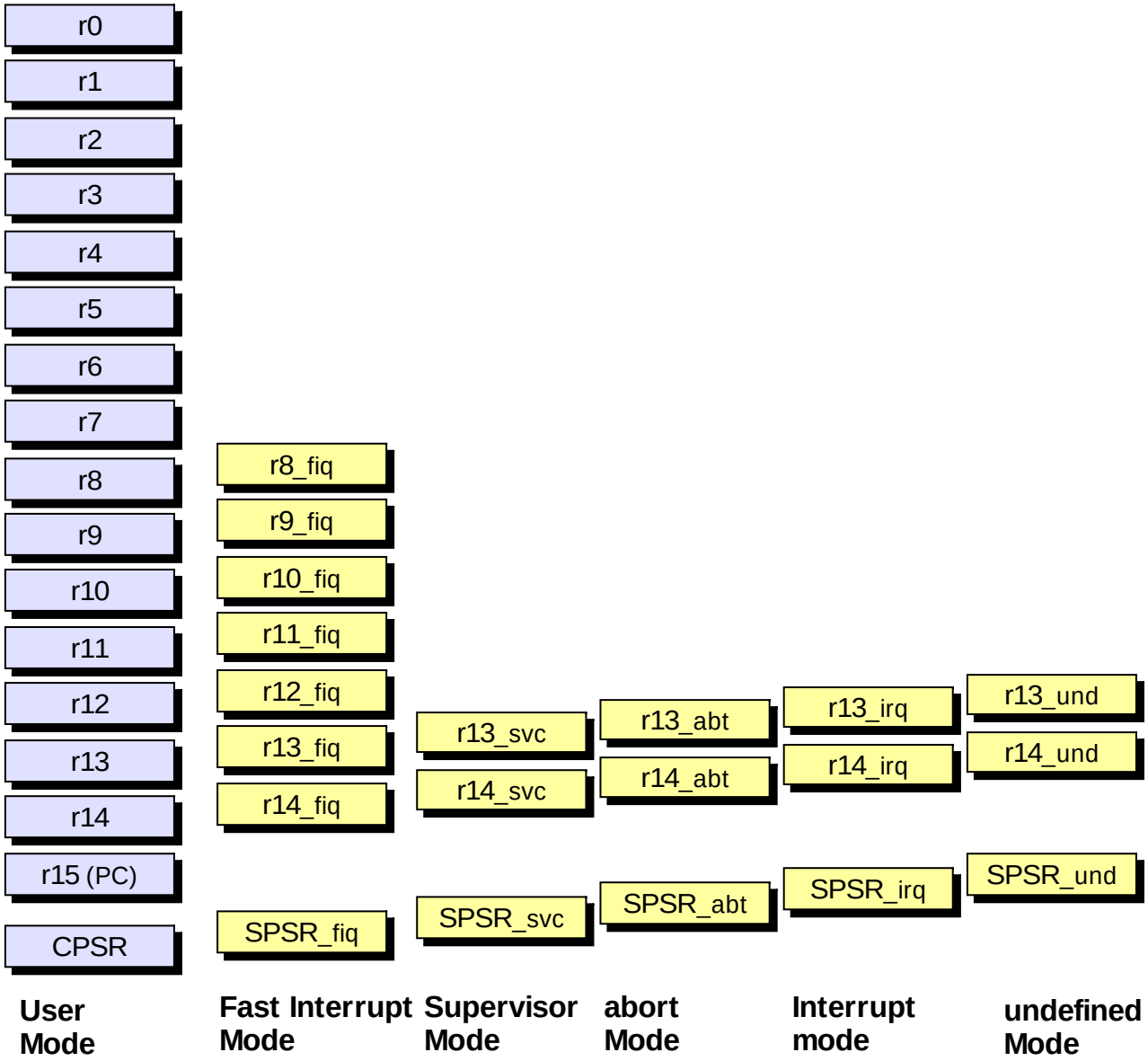
Add d, s1; d = d + s1



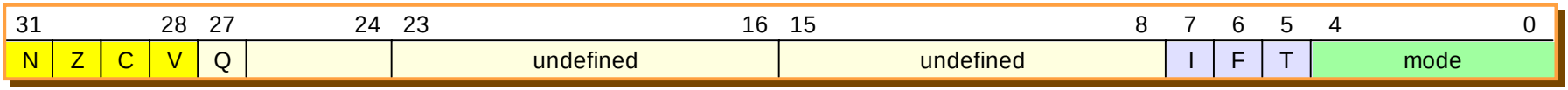
1 Adress Instruktionsformat (Intel: manche Befehle, MU0)

Add s1; accumulator = accumulator + s1

ARM Registerstruktur



Current Program Status Register (CPSR)



- **Condition Code Flags**
  - N = Negatives ALU Ergebnis
  - Z = Alu Ergebnis ist Null
  - C = Alu Ergebnis erzeugte Carry
  - V = Alu erzeugte Overflow
- Interrupt Disable Bits
  - I = 1, disables IRQ
  - F = 1, disables FIQ

- T Bit
  - T = 0, Prozessor in Arm State
  - T = 1, Prozessor in Thumb State
- Mode Bits
  - Spezifizieren den Prozessor Mode

Zugriff auf Daten

- RISC Prozessoren können keine direkten Zugriffe auf Daten durchführen (Die Adresse passt nicht in den Befehlscode)
- Die Adresse muss daher vor dem Zugriff in ein Register geladen werden.
- Das Laden der Adresse erfolgt normalerweise über eine Konstante, die PC relativ adressiert wird
- Der Befehl

  ldr Register, Label
  - wird dabei vom Assembler in einen Befehl

  ldr Register, [pc + Offset]
  - umgesetzt

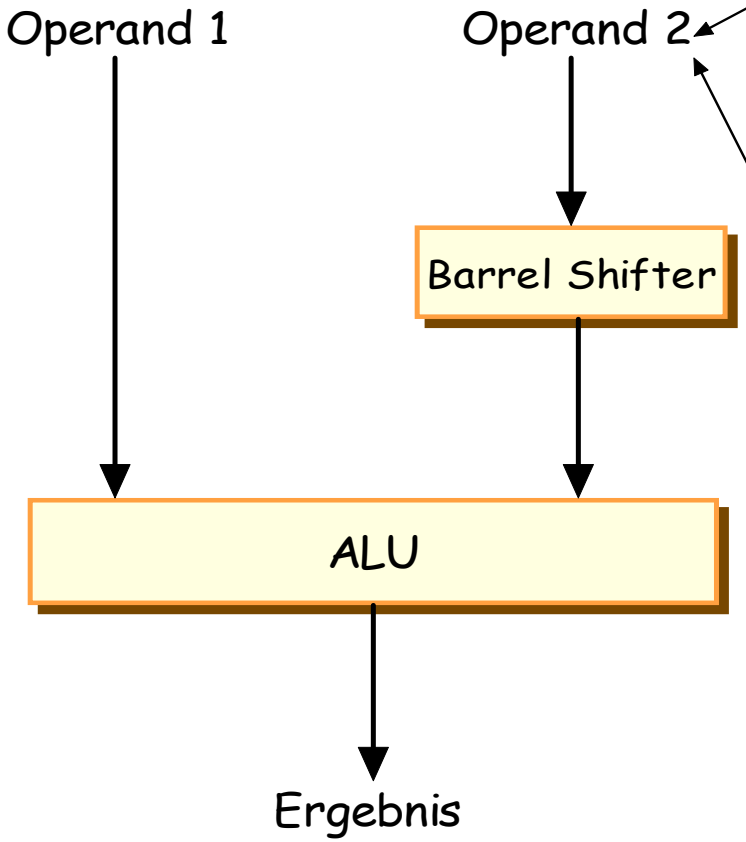
```
    ldr    r2, .L2   ; r2 = ADR(y)
    ldr    r3, .L2+4 ; r3 = ADR(x)
    ldr    ip, [r3, #0] ; ip = x
    ldr    r3, [r2, #0] ; r3 = y
    mov r4, #0x40    ; r4 = 64
    str    r4, [r2]       ; y = 64
...
    .align          2
.L2:
    .word           y
    .word           x
    .word           z
    .word           .LC0
```

Arithmetische und logische Befehle

- Die Arithmetischen Befehle der ARM Prozessoren sind 3 Register Befehle
- Im Befehl werden das Zielregister und die beiden Operanden angegeben
- Beispiel
  - add r0, r0, r3
- Die Bedingungsbits des Statusregisters werden nur bei den Test und Compare Befehlen automatisch gesetzt.
- Alle anderen Befehle müssen die Statusbits explizit setzen
- Beispiel
  - subs r0, r0, #1

Befehl	Operation
AND	Rd:= Op1 AND Op2
EOR	Rd:= Op1 EOR Op2
SUB	Rd:= Op1 - Op2
RSB	Rd:= Op2 - Op1
ADD	Rd:= Op1 + Op2
ADC	Rd:= Op1 + Op2 + C
SBC	Rd:= Op1 - Op2 + C - 1
RSC	Rd:= Op2 - Op1 + C - 1
TST	set condition codes on Op1 AND Op2
TEQ	set condition codes on Op1 EOR Op2
CMP	set condition codes on Op1 - Op2
CMN	set condition codes on Op1 + Op2
ORR	Rd:= Op1 OR Op2
MOV	Rd:= Op2
BIC	Rd:= Op1 AND NOT Op2
MVN	Rd:= NOT Op2

Benutzung des Barrel Shifters



Register, Optional kann eine Shift Operation angwandt werden

Shift Werte können sein

- 5 Bit unsigned Integer
- Wert in einem anderen Register

Wird benutzt zur Multiplikation mit Konstanten

Immediate Wert + Shift Operation

- 8 Bit Zahl im Bereich zwischen 0 und 255
- kann nach links oder rechts verschoben werde

```
ADD     r3, r2, r1, LSL #3          ; r3 := r2 + 8 x r1
```

```
ADD     r5, r5, r3, LSL r2          ; r5 := r5 + r3 x 2**r2
```

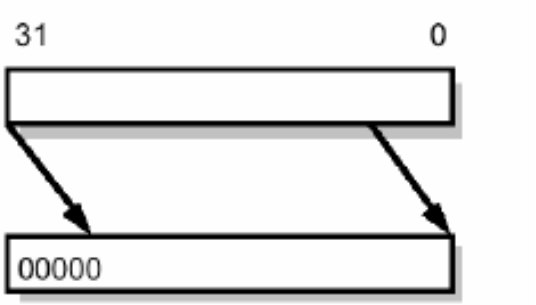
Der Barrel Shifter

LSL: Logical Shift Left

Multiplikation mit einer Zweier Potenz

LSR: Logical Shift Right

Division durch eine Zweier Potenz

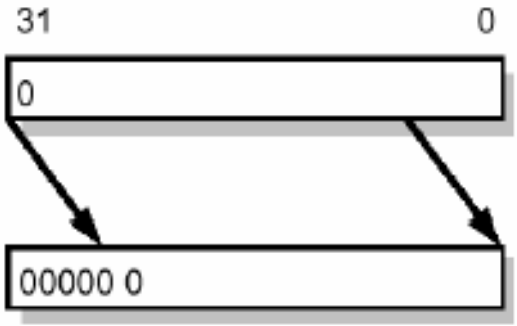


Der Barrel Shifter

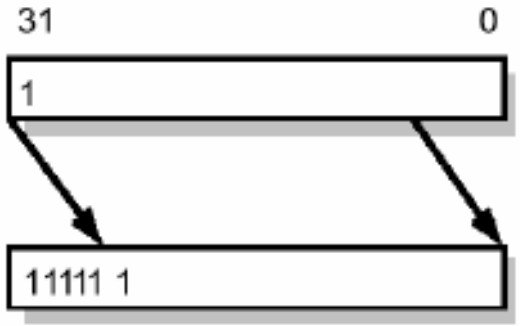
ASR: Arithmetic Shift Right

Division durch eine Zweier Potenz
unter Erhaltung des Vorzeichens

ASL = LSL



ASR #5, positive operand



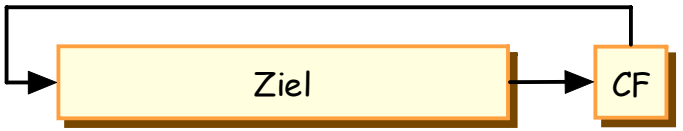
ASR #5, negative operand

Der Barrel Shifter

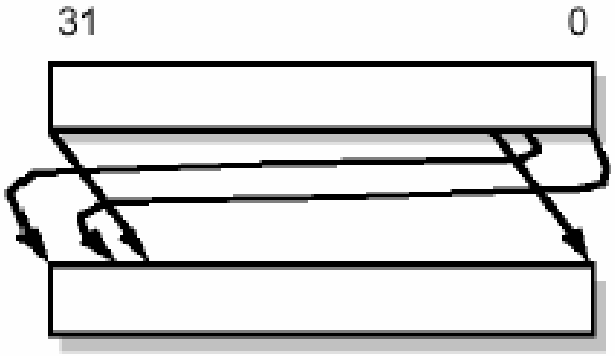
ROR: Rotate Right

Bit Rotation

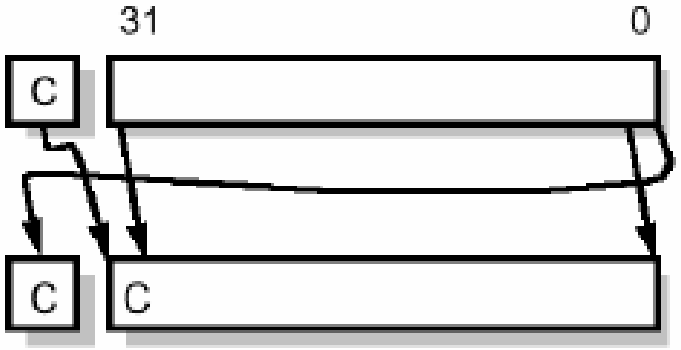
RRX: Rotate Right Extended



Bit Rotation durch das Carry Flag hindurch



ROR #5



RRX

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 mit 5
  - add r0, r1, r1, LSL #2
- Durch Zusammensetzen mehrerer Instruktionen können auch komplexere Multiplikationen durchgeführt werden
- Beispiel: Multiplikation mit 10
  - add r0, r1, r1, LSL #2
  - mov r0, r0, LSL #1
- Beipiel Multiplikation mit 119 = 17 * 7 = (16 + 1) * (8 - 1)
  - add r2, r3, r3, LSL #4 // r2 = r3 * 17
  - rsb r2, r2, r2, LSL #3 // r2 = r2 * 7

Grundsätzliches zu ARM

Drei Befehlstypen

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

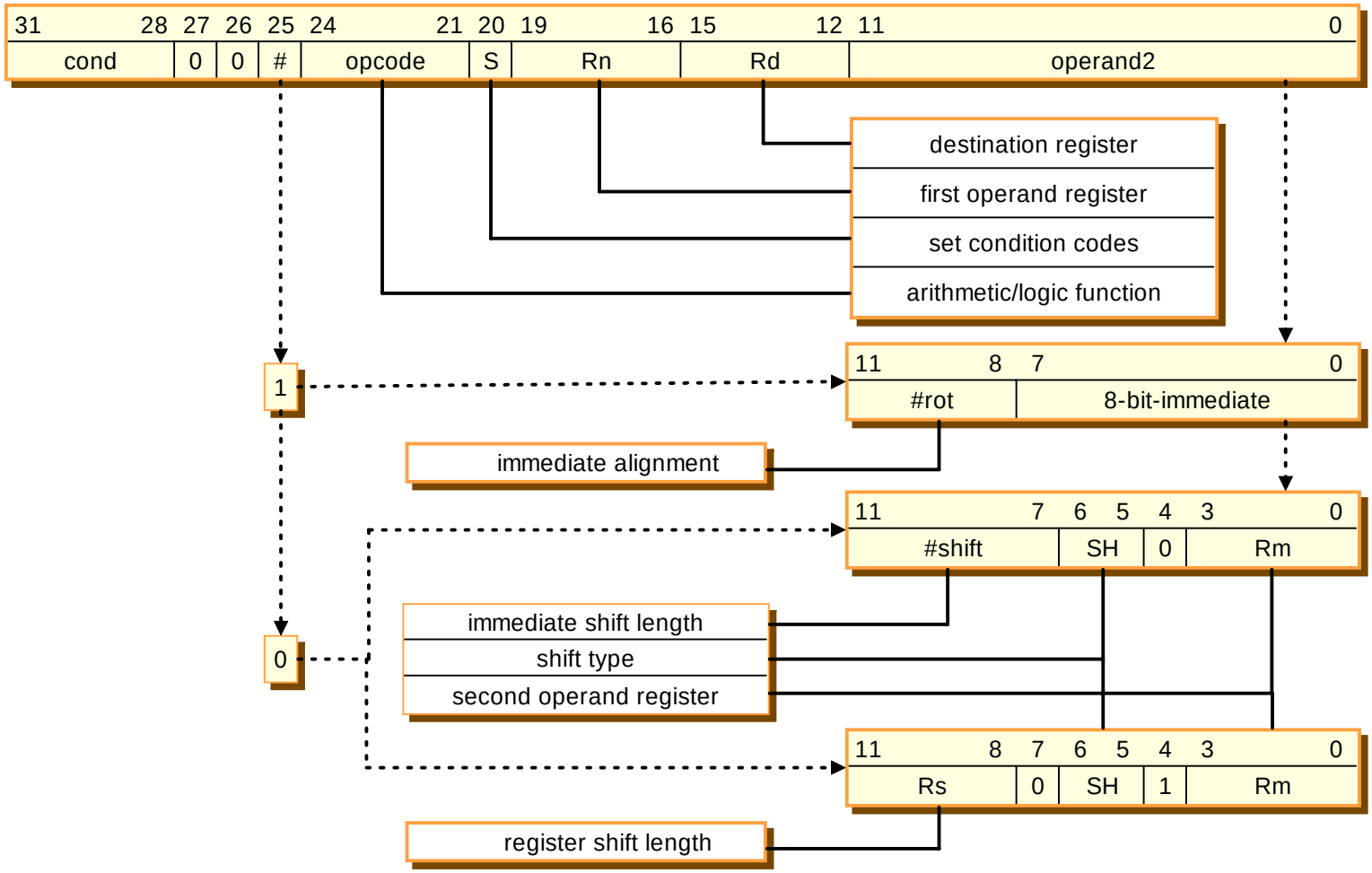
Regeln:

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

Beispiel:

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

Arithmetische und logische Befehle



Operation Codes - Arithmetik

Opcode	Mnemonic	Operation
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

bedingte Ausführung möglich

Bedingte Ausführung

Field mnemonic	Condition
EQ	Z set (equal)
NE	Z clear (not equal)
CS/HS	C set (unsigned >=)
CC/LO	C clear (unsigned <)
MI	N set (negative)
PL	N clear (positive or zero)
VS	V set (overflow)
VC	V clear (no overflow)
HI	C set and Z clear (unsigned >)
LS	C clear and Z set (unsigned <=)
GE	N and V the same (signed >=)
LT	N and V differ (signed <)
GT	Z clear, N and V the same (signed >)
LE	Z set, N and V differ (signed <=)
AL	Always execute (the default if none is specified)

Sprungbefehle

- Sprungbefehle dienen dazu den Kontrollfluss von Programmen zu kontrollieren
- In der Kombination mit Vergleichsbefehlen können mit Ihnen alle wichtigen Kontrollflussanweisungen wie „if.. else“, „while“, „for“ oder „switch“ realisiert werden.
- Das Ziel einer Sprunganweisung ist immer eine Marke (engl. Label).
- Beispiel:

```
Marke1   CMP R0, #0
BNE      Marke1
         .....
         .....
         .....
         .....
Marke1   MOV R5, #5
         .....
```

Branch	Interpretation
B	ohne Bedingung
BAL	always
BEQ	equal
BNE	not equal
BPL	plus
BMI	minus
BCC	carry clear
BLO	lower
BCS	carry set
BHS	higher or same
BVC	overflow clear
BVS	overflow set
BGT	greater than
BGE	greater or equal
BLT	less than
BLE	less or equal
BHI	higher
BLS	lower or same

Bedingte Ausführung am Bsp. von branch

Branch	Interpretation
B	ohne Bedingung
BAL	always
BEQ	equal
BNE	not equal
BPL	plus
BMI	minus
BCC	carry clear
BLO	lower
BCS	carry set
BHS	higher or same
BVC	overflow clear
BVS	overflow set
BGT	greater than
BGE	greater or equal
BLT	less than
BLE	less or equal
BHI	higher
BLS	lower or same

Operand 2 als Konstante

Befehlsformat für Datenverarbeitungsbefehle reserviert 12 Bits für Operand 2

ergäbe einen Wertebereich von max. 2^12 = 4096

aufgeteilt in Wertebereich mit 8 Bits (0 – 255)

Diese 8 Bits können (rechts) rotiert werden mit einer geraden Anzahl von Positionen (ROR mit 0, 2, 4,..30).

Ergibt eine weitaus grössere Abdeckung des Zahlenbereiches.

Einige Konstanten müssen dennoch vom Speicher geladen werden.

Operand 2 als Konstante

Wertebereich:

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

Beispiel MOV - Befehl:

```
MOV r0, #0xFF000000        ; => MOV r0, #0xFF, 8
```

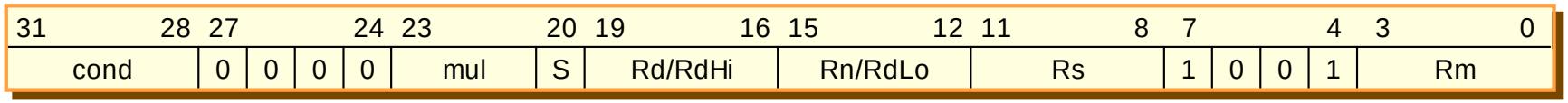
Beispiel MVN: erzeugt das bitweise Komplement (1er Komplement)

```
MOV r0, #0xFFFFFFFF        ; umgesetzt zu  MVN r0, #0
```

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

Beispiel:mov.s

Multiplikation



Assembler Format

MUL{cond}{S} Rd, Rm, Rs

MLA{cond}{S} Rd, Rm, Rs, Rn

<mul>{cond}{S} RdHi, RdLo, Rm, Rs

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

Übung 1

1. Welche Befehle implementieren folg. Zuweisungen:
   a) r0 = 16
   b) r1 = r0 * 4
   c) r0 = r1 / 16 (r1 2's Kompl.mit Vorz.)
   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
```

Datentransfer Befehle

3 Arten von Datentransfer Befehle

- Einzeltransfer Load/Store Befehle
- Blocktransfer Load/Store Befehle
- Einzelregister Transfer Befehl (MRS, MSR)

Verschieden Möglichkeiten der Adressierung

Immediate Adressierung (z.B. MOV R1, #4)
Register Adressierung (z.B. ADD R1, R2, R3)
Register Indirekte Adressierung
Laden von Adressen in Register
Register Indizierte Adressierung

Befehle für LOAD und STORE

LDR = **L**oa**D** **R**egister
STR = **ST**ore **R**egister

Adressierung: Immediate Adressierung

- Bei der Immediate Adressierung wird ein Operand direkt im Befehl gespeichert

- **ARM:**

```
mov R0, #8          ; Lade in das Register R0 den
                    ; Wert 8
```

- Bei Intel Prozessoren kann der Immediate Wert eine Größe von 32 Bit besitzen
- Bei den meisten Risc Prozessoren ist der Immediate Wert deutlich kleiner (12 Bit bei ARM)
- Da der Wert im Befehl enthalten ist, ist für immediate Adressierung kein weiterer Speicherzugriff erforderlich

Direkte Adressierung

- Die Adresse von der ein Wert geladen werden soll, befindet sich im Opcode.
- Nachteil: die Adresse kann zur Laufzeit nicht mehr geändert werden
- Risc Prozessoren können keine 32 Bit Adressen in ihrem Opcode unterbringen, daher beherrschen sie diese Adressierungsart nicht
- Beispiel Intel

```
        mov   ax, hugo
...
...
hugo      dw    ?
```

Register Adressierung

- Register Adressierung ist vom Konzept identisch zur direkten Adressierung, nur daß jetzt statt der Speicheradresse eine Registeradresse benutzt wird.
- Intel

```
add     ax, bx
```

- **ARM**

```
add     r0, r1, r2
```

- In Load Store Architekturen nutzen alle Befehle diese Adressierungsart, außer den Load und Store Befehlen selbst. Aber auch bei diesen Befehlen ist das Ziel oder die Quelle ein Register

Register Indirekte Adressierung

- In dieser Adressierungsart kommt auch einer der Operanden aus dem Speicher, aber die Adresse des Speichers steht in einem Register. (Adressierung über Pointer)
- Der Vorteil dieser Adressierungsart ist, daß man den gesamten Speicherbereich adressieren kann, ohne daß man eine Adresse in der Instruktion unterbringen muß.
- Intel

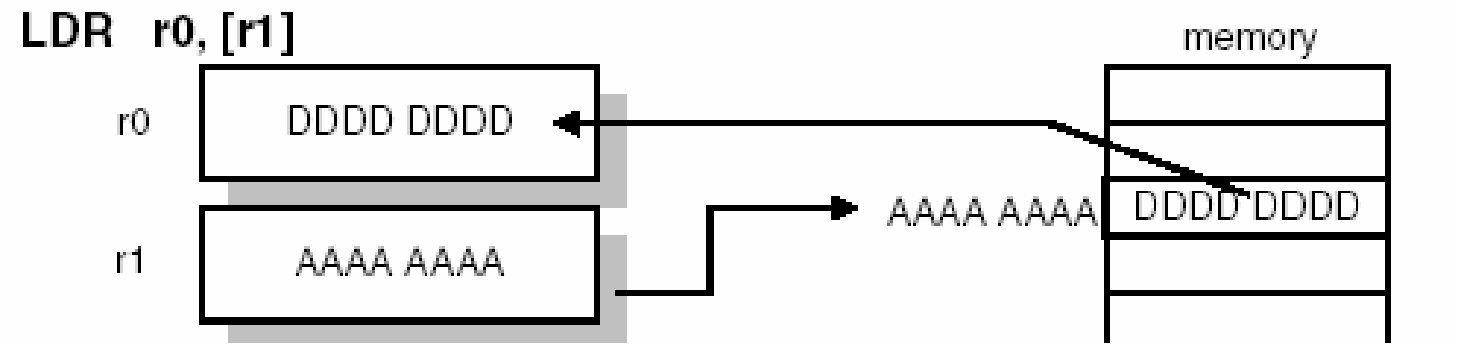
  mov ax, [bx]
- **ARM**

  ldr r0, [r1]
- Diese Adressierungsart ist in Load und Store Architekturen nur in den Load und Store Befehlen erlaubt

Register Indirekte Adressierung (2)

Benutzt einen Registerwert (Basisregister) als Speicheradresse zum Laden oder Speichern des Wertes an dieser Adresse

```
LDR     r0, [r1]        ; r0 := mem32[r1]
STR     r0, [r1]        ; mem32[r1] := r0
```



Wie kann Basisregister gesetzt werden ? Initialisierung ?

Laden von Adressen in Register

- Das Laden einer 32 Bit Adresse in ein Register erfolgt mit Hilfe von Pseudoinstruktionen, die der Assembler mit Hilfe eines Satzes von Regeln auf bestmögliche Art abbildet

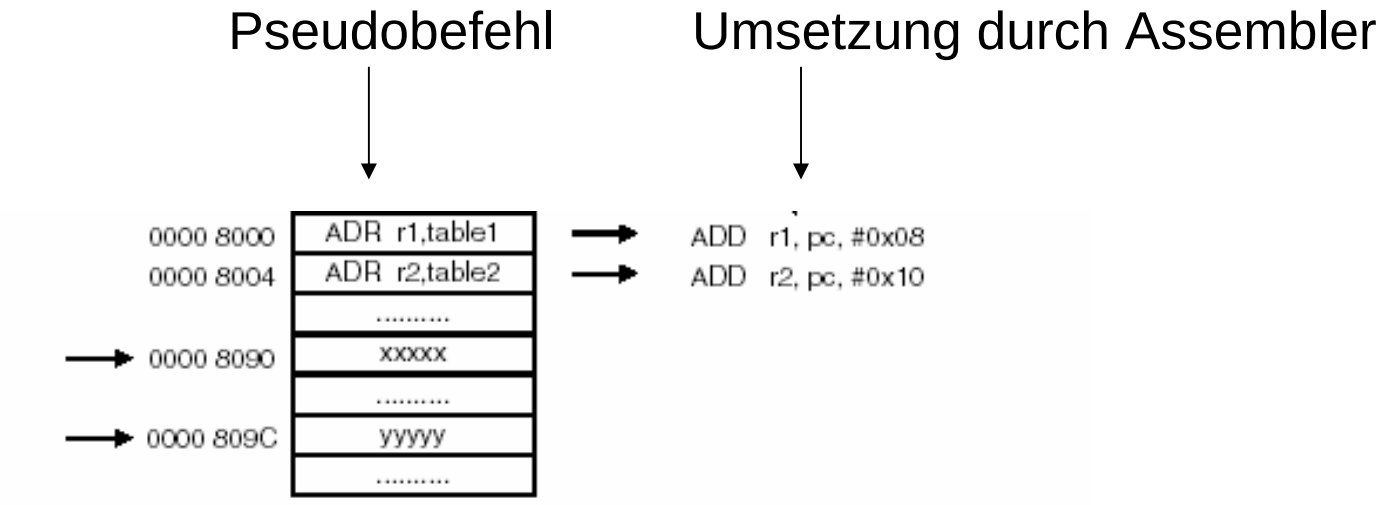
```
        ADR  R1, Table1

....

...

Table1
```

- Die Umsetzung des ADR Befehls erfolgt vorzugsweise durch einen Offset zum Program Counter (PC=R15). Der Offset kann bis zu 1 KB betragen.

Laden von Adressen in Register (2)

Beispiel: Kopieren von Daten aus Tabelle 1 in Tabelle 2

```
copy        ADR   r1, TABLE1     ; r1 points to TABLE1
            ADR   r2, TABLE2     ; r2 points to TABLE2
            LDR   r0, [r1]       ; load first value ....
            STR   r0, [r2]       ;   and store it in TABLE2
            .......
TABLE1      ......               ; <source of data>
            ......
TABLE2      ......               ; <destination of data>
```



Laden von Adressen in Register (3)

Weiterentwicklung des Beispiels: Kopieren von Daten

```
copy    ADR   r1, TABLE1     ; r1 points to TABLE1
        ADR   r2, TABLE2     ; r2 points to TABLE2
        LDR   r0, [r1]       ; load first value ....
        STR   r0, [r2]       ;    and store it in TABLE2
        ADD   r1, r1, #4     ; step r1 onto next word
        ADD   r2, r2, #4     ; step r2 onto next word
        LDR   r0, [r1]       ; load second value ...
        STR   r0, [r2]       ;    and store it
        ......
```

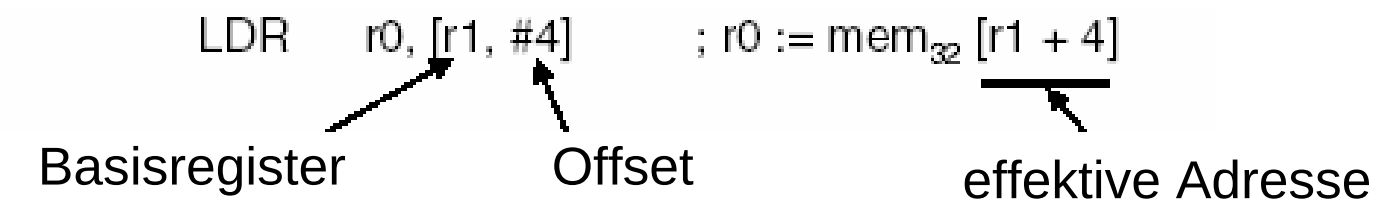
Indizierte Adressierung

- Es kommt sehr häufig vor, daß auf Speicher zugegriffen wird, der einen Offset relativ zu einer Basisadresse besitzt
- Adressierung über ein Register plus einem konstanten Offset wird indexed adressing genannt
- Beispiele:
  - Zugriff auf Strukturen
  - Zugriff in Arrays

- **ARM**

```
ldr r0, [r1, #4]      ; pre indexed
ldr r0, [r1, #4]!     ; pre indexed mit Autoinkrement
ldr r0, [r1], #4      ; post indexed
```

Beispiel: **pre-indexed** (ändert Registerwert nicht !)



Indizierte Adressierung (2)

Optimierung des Beispiels: Kopieren von Daten

```
copy    ADR   r1, TABLE1      ; r1 points to TABLE1
        ADR   r2, TABLE2      ; r2 points to TABLE2
        LDR   r0, [r1]        ; load first value ....
        STR   r0, [r2]        ;   and store it in TABLE2
        LDR   r0, [r1, #4]    ; load second value ...
        STR   r0, [r2, #4]    ;   and store it
        ......
```

Indizierte Adressierung (3)

- **ARM**

```
ldr r0, [r1, #4]      ; pre indexed
ldr r0, [r1, #4]!     ; pre indexed mit Autoinkrement
ldr r0, [r1], #4      ; post indexed
```

Beispiel: **pre-indexed mit Autoinkrement** (ändert Registerwert !)

```
LDR    r0, [r1, #4]!     ; r0 : = mem32 [r1 + 4]
                         ; r1 := r1 + 4
```

! gibt an, dass Basisregister verändert wird

Beispiel: **post-indexed** (ändert Registerwert !)

```
LDR    r0, [r1], #4      ; r0 : = mem32 [r1]
                         ; r1 := r1 + 4
```

Zieladresse in R1, danach wird R1 inkrementiert

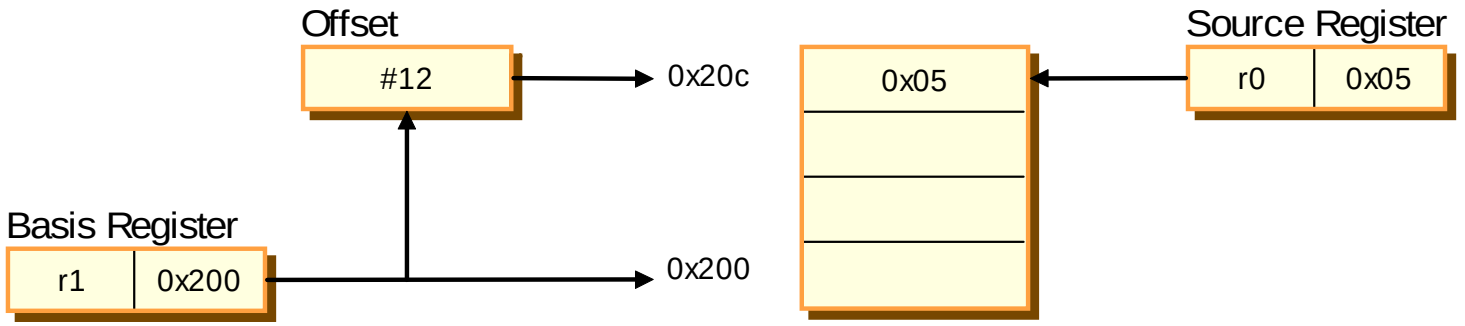
Indizierte Adressierung (4)

Optimierung des Beispiels: Kopieren von Daten

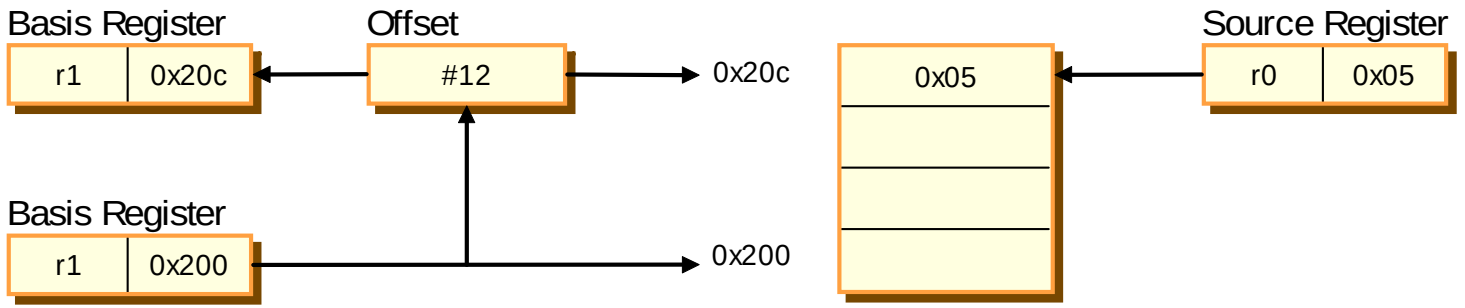
```
copy        ADR    r1, TABLE1      ; r1 points to TABLE1
            ADR    r2, TABLE2      ; r2 points to TABLE2
loop        LDR    r0, [r1], #4    ; get TABLE1 1st word ....
            STR    r0, [r2], #4    ;    copy it to TABLE2
            ???                    ; if more, go back to loop
            ......
TABLE1      ......                 ;  < source of data >
```

Indizierte Adressierung (5)

Pre Indexed: str r0, [r1, #12]

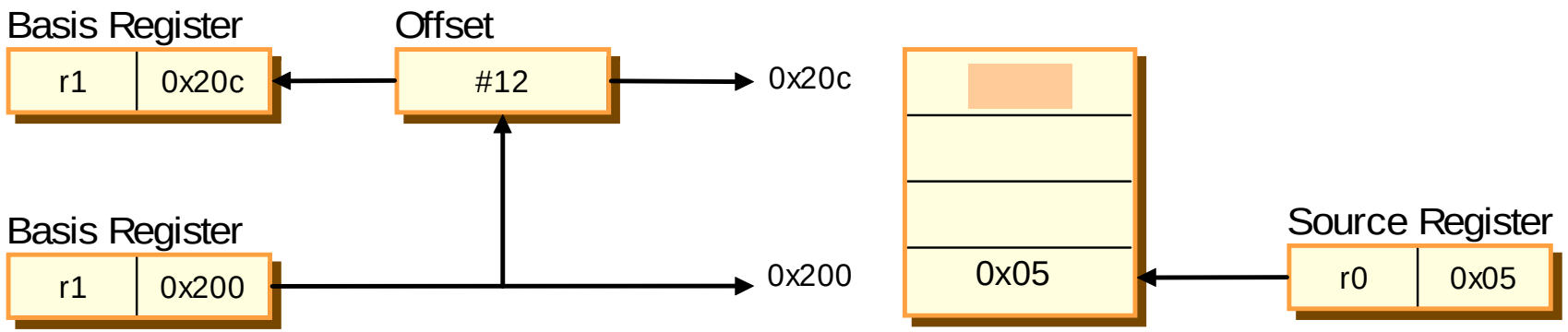


Pre Indexed mit Auto Update: str r0, [r1, #12]!



Indizierte Adressierung (6)

Post Indexed mit Auto Update: str r0, [r1], #12



Basis indizierte Adressierung

- Moderne Prozessoren besitzen häufig noch eine Adressierungsart in der zu dem Indexregister noch ein Basisregister addiert wird
- Anwendungen
  - Vektoradressierung
  - Stackadressierung
- ARM

```
ldr r1,[r2, r0, LSL #2]
```

Die Befehle LDR und STR

- Die Befehle LDR und STR dienen zum Laden und speichern von Daten
- Der Zugriff erfolgt indirekt, d.h. immer relativ zu einem Register
- Die Speicherung kann auch inidziert und mit einem Offset vorgenommen werden
- Der zum Zugriff genutzte Zeiger kann vor oder nach dem Zugriff inkrementiert oder dekrementiert werden

```
LDR | STR {<cond>} {B} Rd, [Rn,
    <offset>] {!}

LDR | STR {<cond>} {B} Rd, [Rn],
    <offset>

LDR | STR {<cond>} {B} Rd, LABEL
          ; PC relativ

MOV     Rd, #value
```

Datenzugriffe (LDR & STR)

- ARM
  - LDR | STR {<cond>} {B} Rd, [Rn, <offset>] {!}
  - LDR | STR {<cond>} {B} Rd, [Rn], <offset>

  - LDR | STR {<cond>} {B} Rd, LABEL ; PC relativ

  - <offset> := # +/-<12 Bit immediate>
  - +/-Rm {, shift }

Datenzugriffe (LDR & STR)

LDR | STR {<cond>} {**B**} Rd, [Rn, <offset>] {!}
LDR | STR {<cond>} {**B**} Rd, [Rn], <offset>

LDR | STR {<cond>} {**B**} Rd, LABEL ; PC relativ

Die Grösse der geladenen (oder gespeicherten) Variablen kann reduziert werden: **B = Byte**

```
LDRB   r0, [r1]          ; r0 := mem8 [r1]
```

Beispiele für LDR und STR

STR R1,[R2,R4]!

Speichert R1 in die Adresse R2+R4 und schreibt diese Adresse nach R2.

STR R1,[R2],R4

Speichert R1 in R2 und schreibt R2+R4 in R2.

LDR R1,[R2,#16]

Lade R1 von der Adresse R2+16, R2 bleibt unverändert.

LDR R1,[R2,R3,LSL#2]

Lade R1 mit dem Inhalt an der Adresse R2+R3*4.

LDREQB R1,[R6,#5]

Bedingtes Laden des Bytes an R6+5 in R1 (Bits 0 bis 7, Bits 8 bis 31= 0).

STR R1,PLACE

Assembler generiert PC relativen Offset zur Addresse PLACE.

...

PLACE

Beispiel für Multiplikation

- Skalar Produkt zweier Vektoren

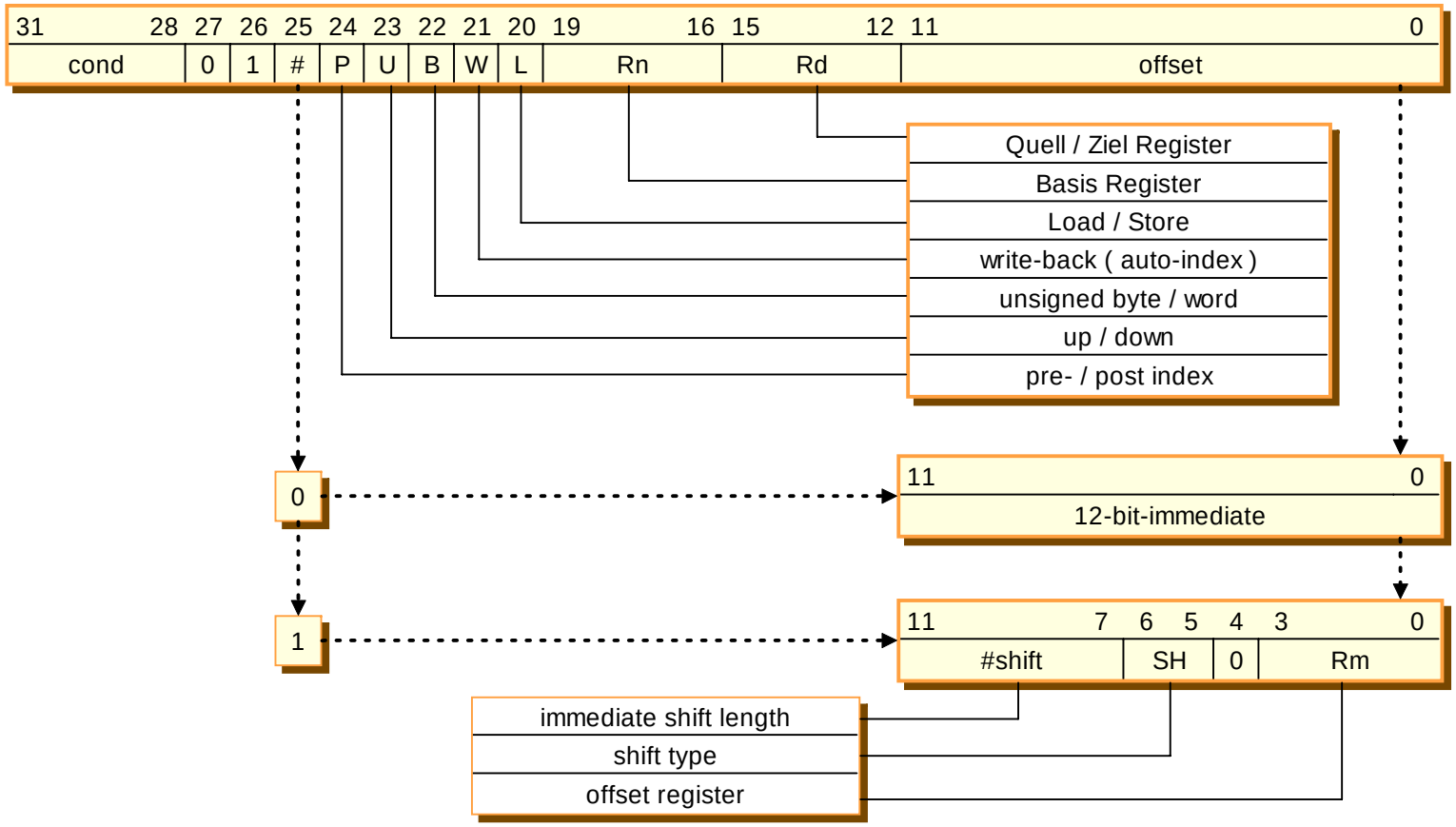
```
        adr     r8, Vektor1     // r8: Zeiger auf Vektor1
        adr     r9, Vektor2     // r9: Zeiger auf Vektor2
        mov     r11, #20        // Dimension des Vektors
        mov     r10, #0         // Initialisierung
    loop
        ....
        ....
        ....
        ....
       bne      loop
```

Beispiel für Multiplikation

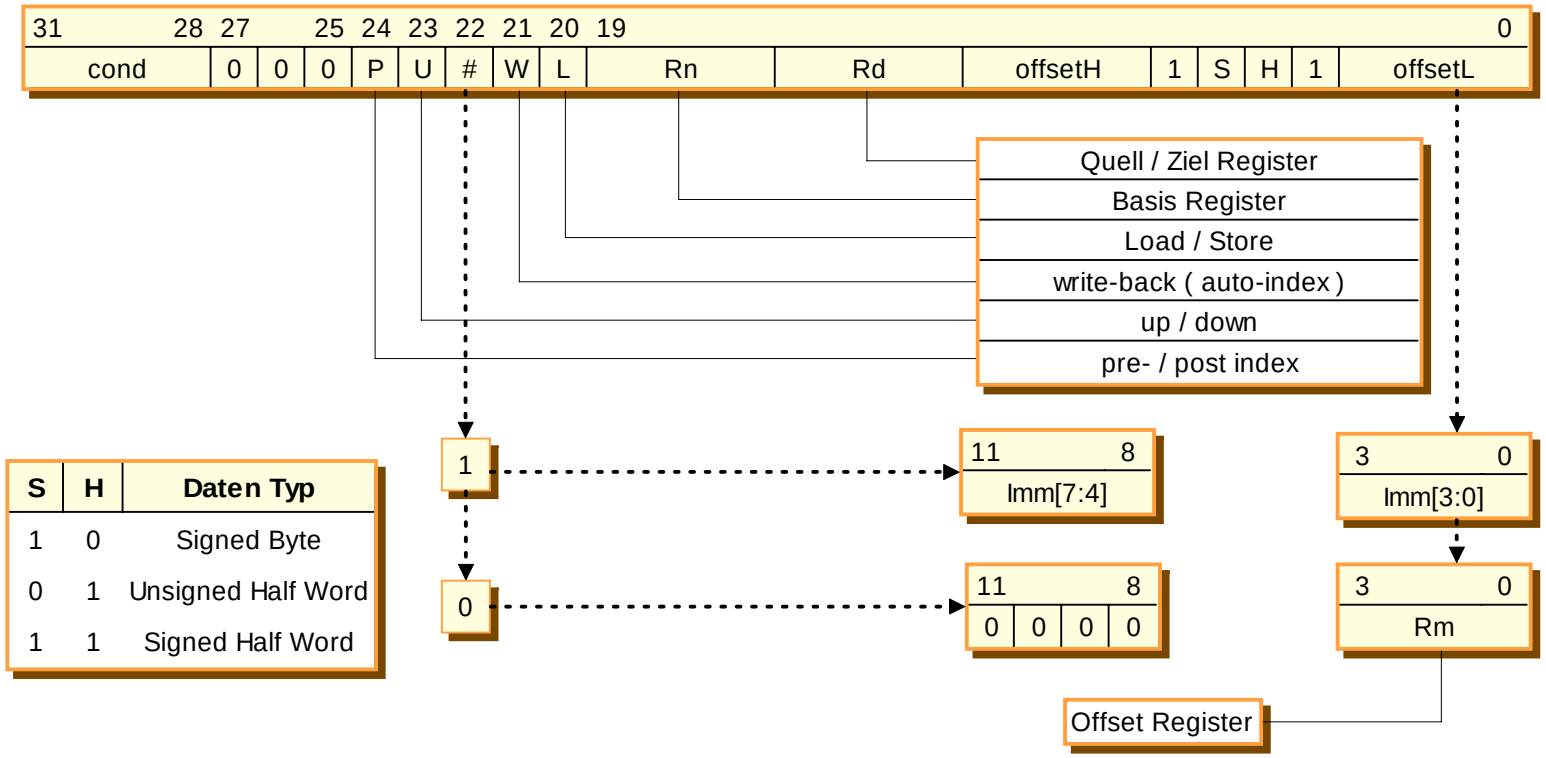
- Skalar Produkt zweier Vektoren

```
        mov   r11, #20
        mov   r10, #0
    loop
        ldr   r0, [r8], #4
        ldr   r1, [r9], #4
        mla   r10, r0, r1, r10     // r10 = r0 * r1 + r10
        subs  r11, r11, #1
        bne   loop
```

Load und Store - Befehlsformat



Load und Store Halbworte



Blocktransfer Befehle

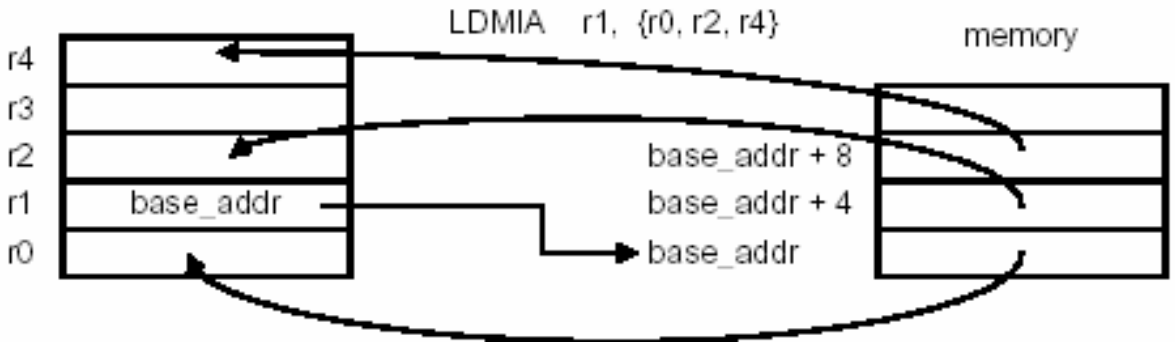
LDR und STR Befehle übertragen jeweils (nur) ein 32 Bit

ARM erlaubt den Transfer von einer beliebigen Teilmenge der 16 Register in einem Befehl

STM **ST**ore **M**ultiple

LDM **L**oa**D** **M**ultiple (Beispiel)

```
LDMIA   r1, {r0, r2, r4}        ; r0 := mem32[r1]
                                ; r2 := mem32[r1+4]
                                ; r4 := mem32[r1+8]
```



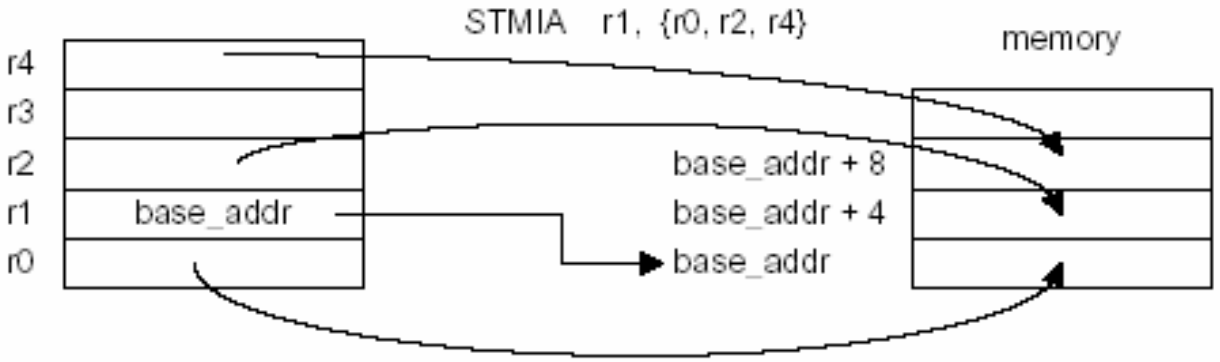
Blocktransfer Befehle

Alle 16 Register können benutzt werden

Achtung: Veränderung vom PC erzwingt Programmverzweigung

Beispiel: Store Multiple

```
STMIA   r1, {r0, r2, r4}        ; mem32[r1] := r0
                                ; mem32[r1 + 4] := r2
                                ; mem32[r1 + 8] := r4
```



Blocktransfer Befehle (Beispiel)

Beispiel: Kopieren von 8 Worten

```
        ADR     r0, src_addr      ; initialize src addr
        ADR     r1, dest_addr     ; initialize dest addr
        LDMIA   r0!, {r2-r9}      ; fetch 8 words from mem
                                  ;    ... and update r0 := r0 + 32
        STMIA   r1, {r2-r9}       ; copy 8 words to mem, r1 unchanged
```

Bei der Benutzung von LDMIA oder STMIA Befehlen:

- wird die Basisadresse inkrementiert
- das Inkrementieren erfolgt nach der Benutzung der Adresse

Blocktransfer Befehle

- Die Richtung in die sich der Basisregister bewegt wird durch den Anhang an den Befehl spezifiziert
  - ldm**ia** / stm**ia** : **i**ncrement **a**fter
  - ldm**ib** / stm**ib** : **i**ncrement **b**efore
  - ldm**da** / stm**da** : **d**ecrement **a**fter
  - ldm**db** / stm**db** : **d**ecrement **b**efore
- Beispiel

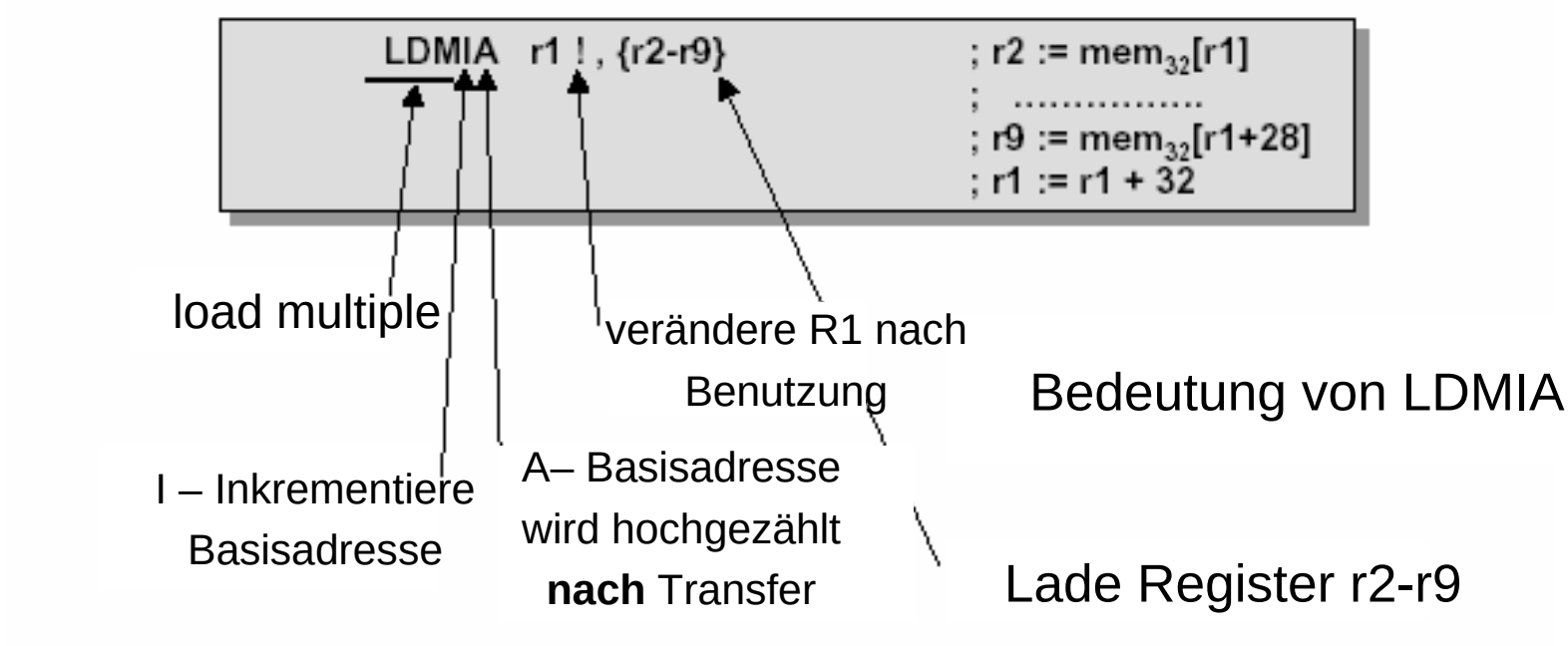
  R12 zeigt auf Quelle, R13 zeigt auf Ziel, R14 zeigt auf Ende

```
loop:   ldmia   r12!, {r0-r11}
        stmia   r13!, {r0-r11}
        cmp     r12, r14
        bne     loop
```

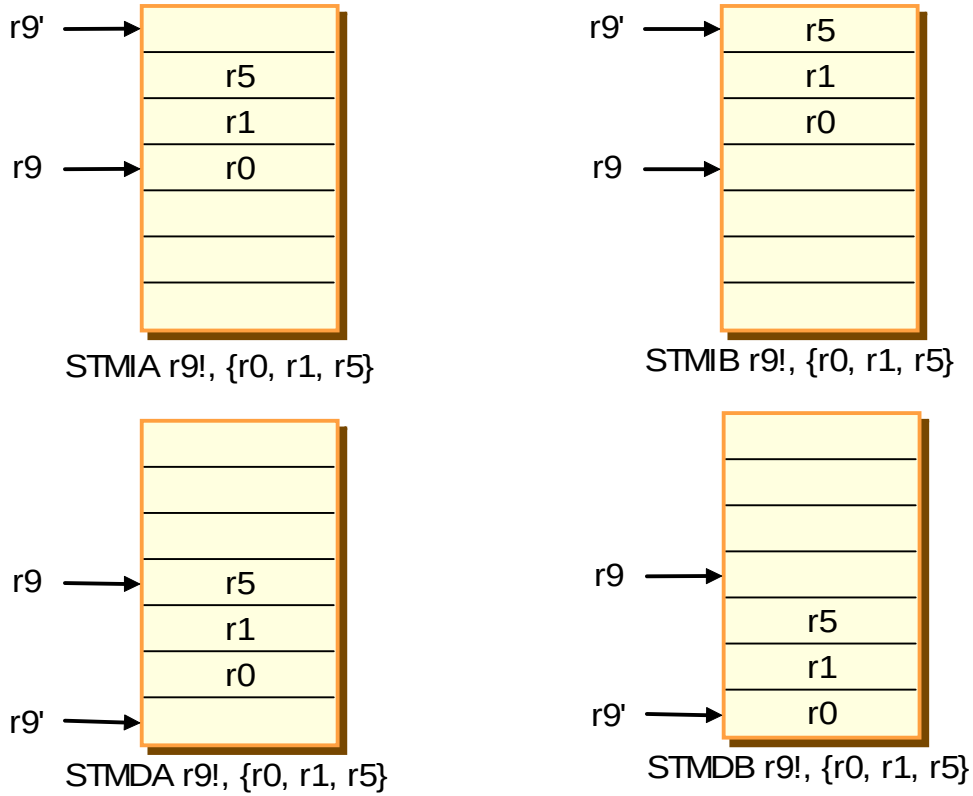
Blocktransfer Befehle

Bisher: Basisregister R1 blieb unverändert

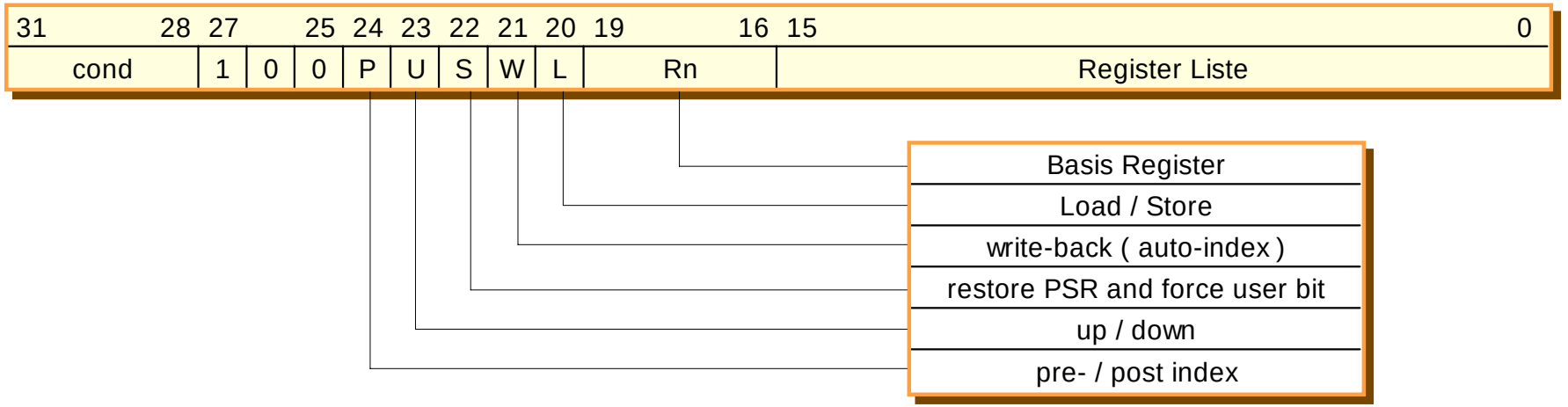
mit ! kann das Basisregister verändert werden



Blocktransfer Befehle



Blocktransfer Befehlsformat

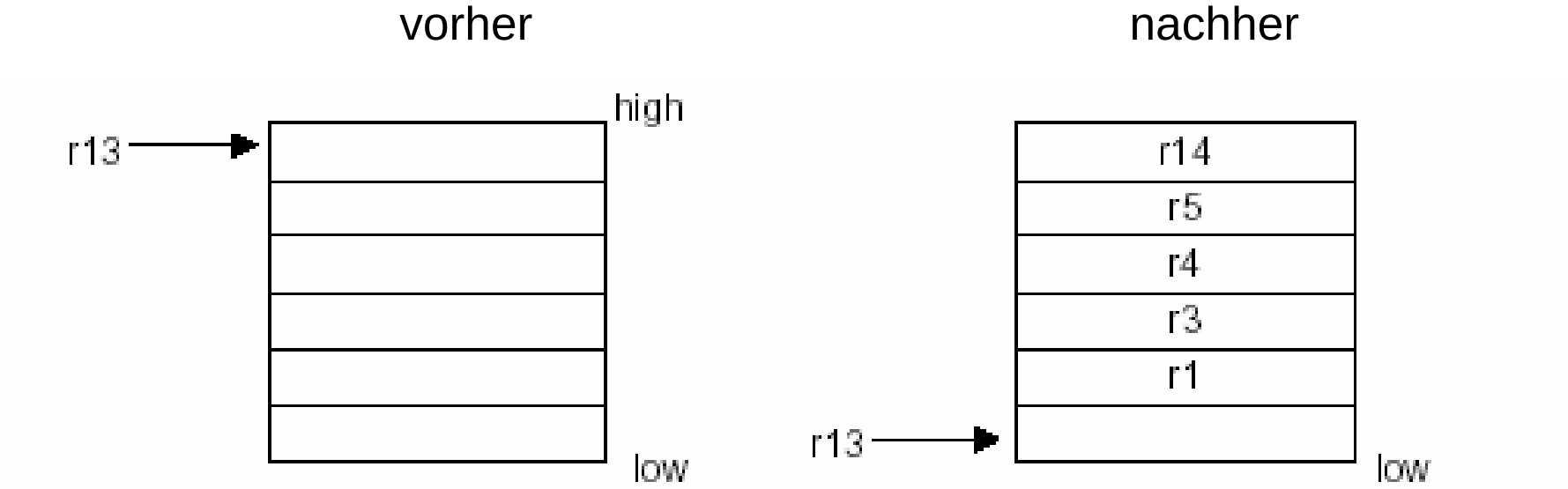


Beispiel: STMIA R13!, {R0-R2, R14} **E8AD 4007**

Stacks

- LDM und STM Befehle können benutzt werden, um einen Last-In First-Out (LIFO) Speicher zu realisieren
- Einen solchen Speicher nennt man STACK (Kellerspeicher)
- Stacks werden benutzt, um temporär Daten zu speichern
- PUSH Befehl, um Daten „auf“ den Stack abzulegen

Beispiel: PUSH {R1, R3-R5, R14}



Stacks

- Eigenschaften des PUSH Befehls
- R13 wird als Adresszeiger benutzt. Diesen nennt man auch Stack Pointer (SP). Vereinbarung !
- Stacks werden benutzt, um temporär Daten zu speichern
- Der Stack wächst „nach unten“ zu kleineren Speicheradressen
- Adresszeiger (= Stackpointer) zeigt auf den ersten freien Speicherplatz
- SP wird nach dem Speicherzugriff dekrementiert

- ARM besitzt keinen PUSH Befehl
- Nutzen der Blocktransferbefehle LDM und STM
- obiges Beispiel:

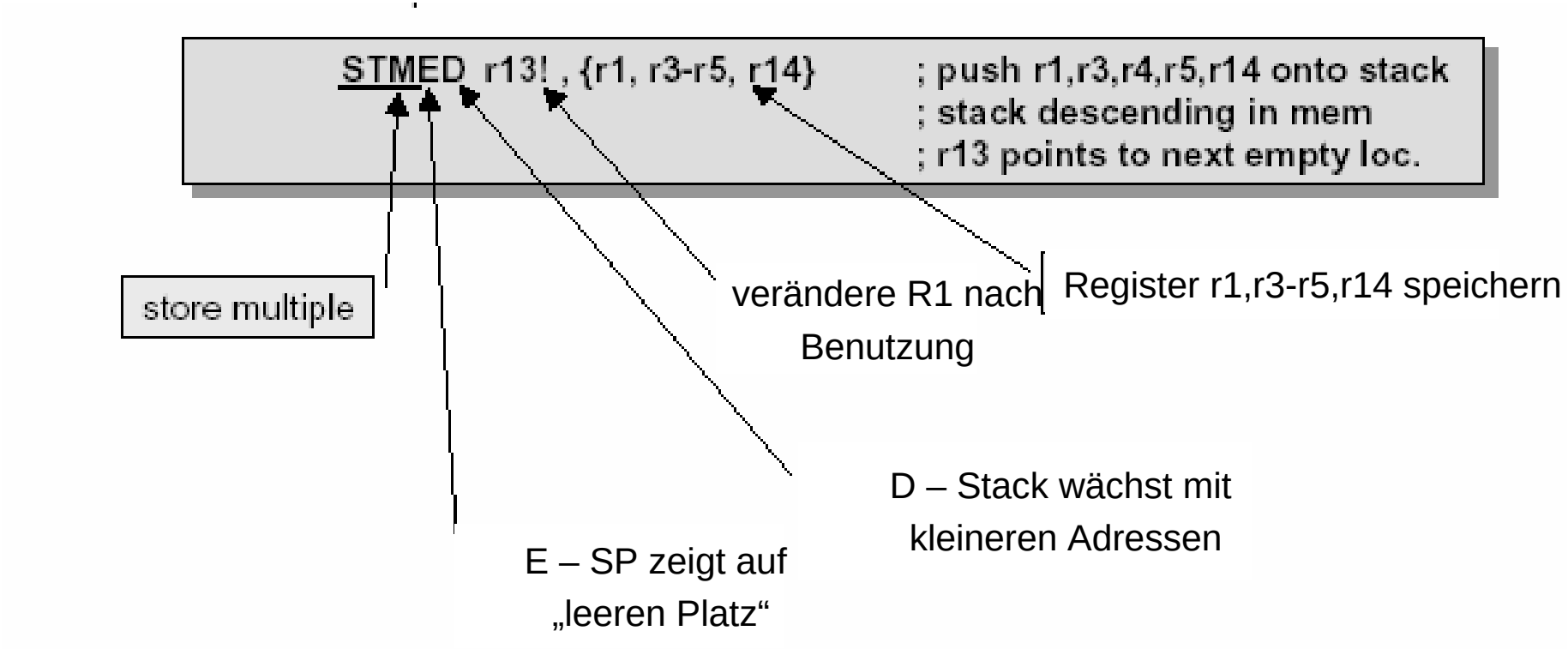
```
STMDA r13!, {r1, r3-r5, r14}      ; Push r1, r3-r5, r14 onto stack
                                  ; Stack grows down in mem
                                  ; r13 points to next empty loc.
```

Stacks

ARM Terminologie

- STM Befehl, der benutzt wird, um einen Stack zu implementieren kann anderen Namen haben

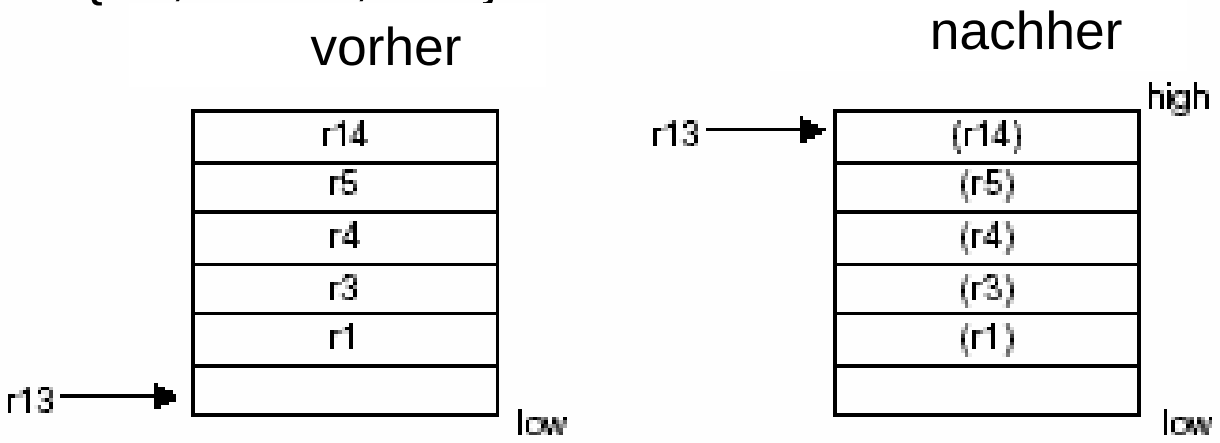
- STMDA ist äquivalent zu STMED



Stacks

- Komplementärer Befehl zu PUSH ist POP
- POP überträgt den Inhalt vom Stack in Register
- POP Befehl gibt es bei ARM nicht

Beispiel: POP {R1, R3-R5, R14}




```
LDMIB   r13!, {r1, r3-r5, r14}        ; Pop r1, r3-r5, r14 from stack
```

äquivalent zu

```
LDMED  r13!, {r1, r3-r5, r14}         ; Pop r1, r3-r5, r14 from stack
```

Stackoperationen

stmfd sp!, { lr }

- Das Register lr wird auf den Stack geschrieben und der Stackpointer dabei nach unten verschoben

ldmfd sp!, { pc }

- Der Program Counter wird vom Stack geholt

stmfd sp!, {REGLIST}

Beispiel:

```
stmfd sp!, {r4, r5, lr}
```

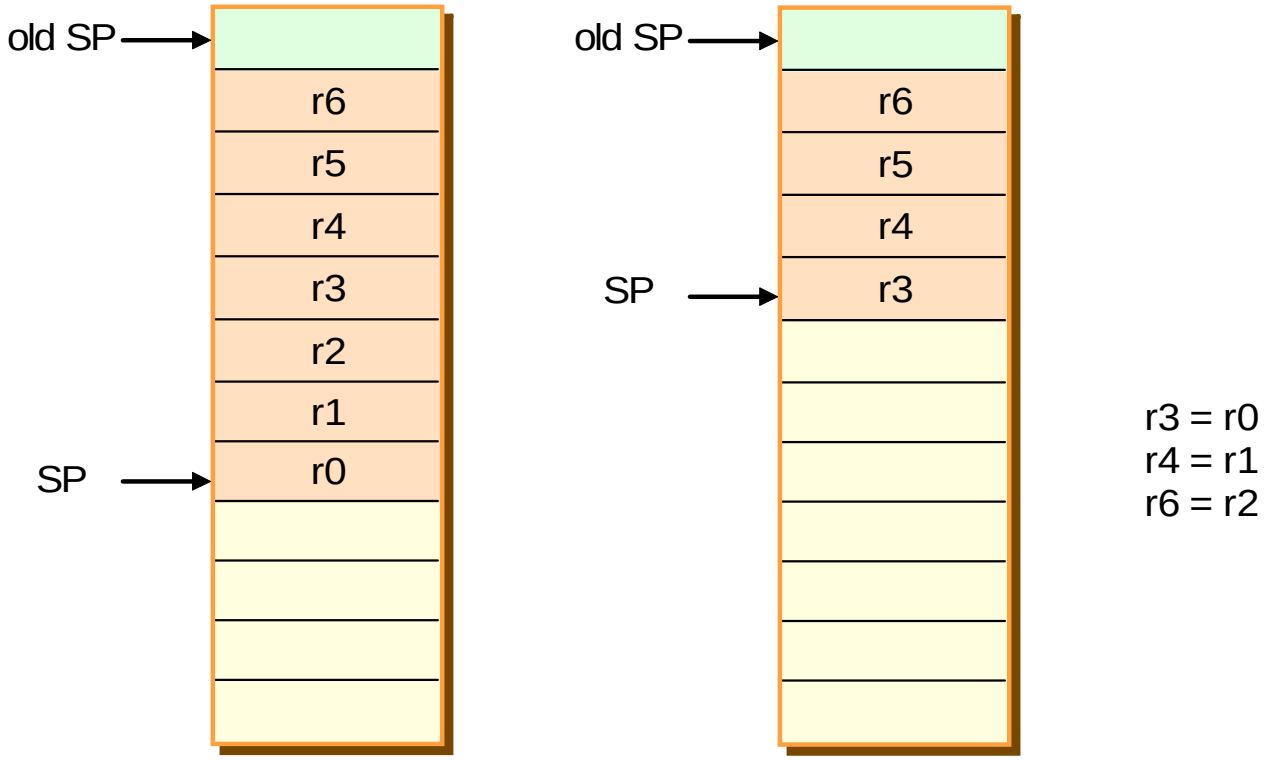
ldmfd sp!, {REGLIST}

Die Liste der Register (REGLIST) wird vom Stack geholt

Beispiel

```
ldmfd sp!, {r4, r5, pc}
```

Stackoperationen



Stack- und Blocktransferbefehle

STACK

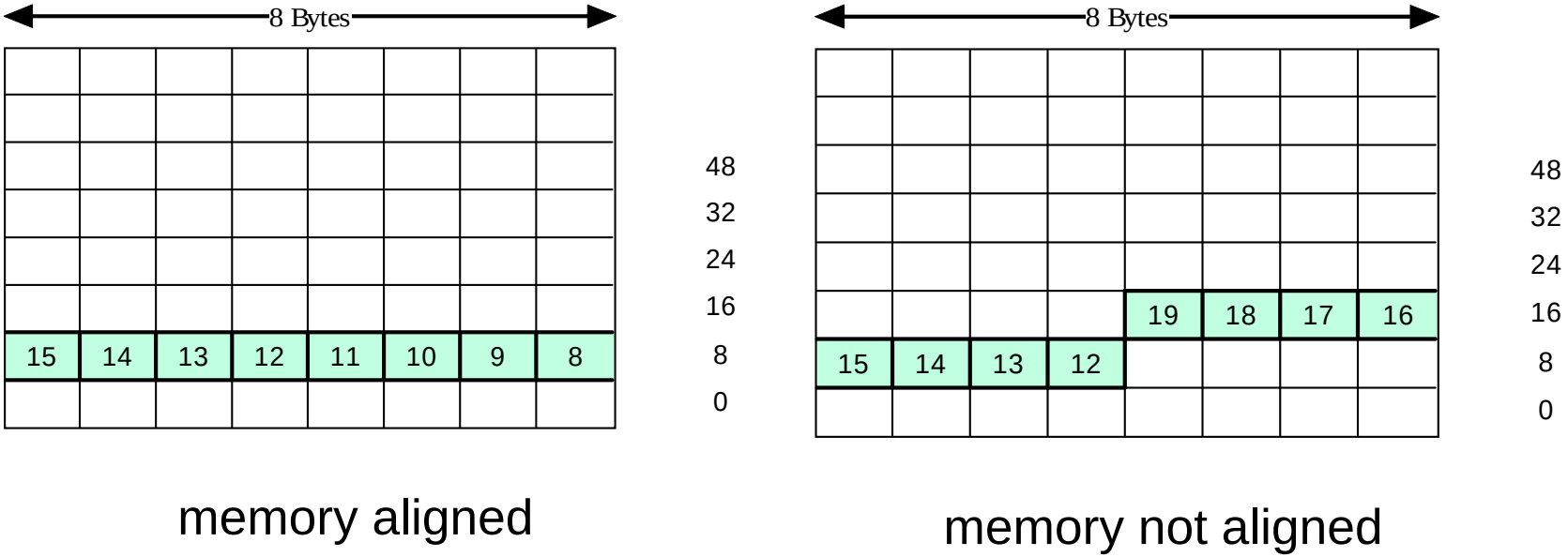
		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Speicheraufteilung

- .text
  - legt eine Textsection an
- .data
  - legt eine Datensection an
- .comm symbol, size
  - legt ein Symbol in die globale bss Section für uninitialisierte Daten
- .global
  - nimmt ein symbol in die globale Symboltabelle auf

- .word Ausdruck
  - legt einen intitialisierten Speicher an
- align Bits
  - sorgt dafür, daß die nachfolgende Anweisung auf einer Speicherstelle steht, deren unterste #Bits 0 sind
- label:
  - legt ein Programmlabel (Marke) an
- .end
  - das Ende des Programms

Speichermodelle



memory aligned

memory not aligned

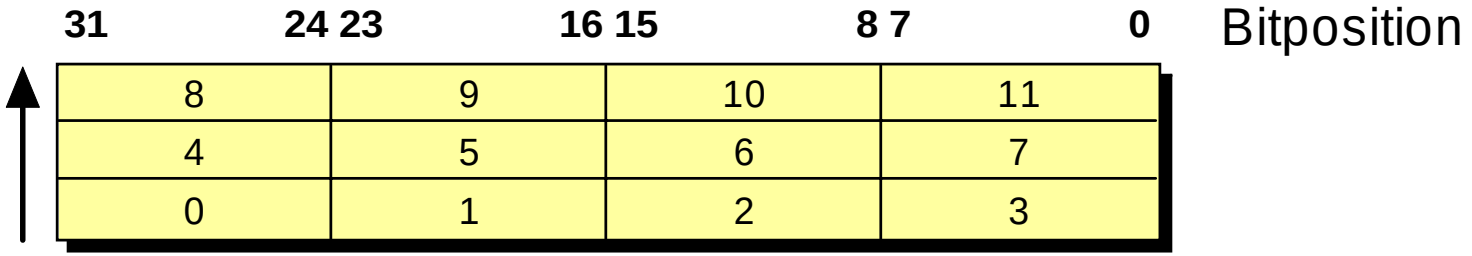
Speichermodelle

- Die meisten Maschinen haben einen linearen Adressraum auf dem ISA Level von 0 bis 2^{32} oder 2^{64} bytes.
- Manche Rechner haben unterschiedliche Adressräume für Instruktionen und Daten.
- Das Lesen eines Wortes, das im letzten Befehl gespeichert wurde stellt für den Entwickler eines Prozessors ein großes Problem dar
  - Pipelines
  - Out of order Execution
  - Multiskalare Prozessoren
  - ➢ Lösung:
    - alle Memory Zugriffe werden serialisiert
    - Zugriff erst nach Sync Befehl korrekt

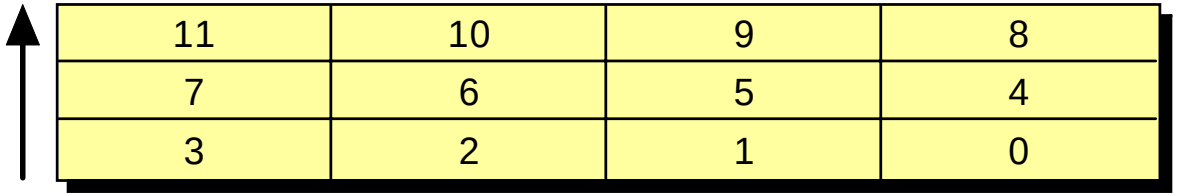
Little und Big Endian

- Der ARM Prozessor kann auf seine Daten in zwei Formaten zugreifen: little endian oder big endian Format.
- Little endian:
  - Least Significant Byte eines Wortes wird an ***Bits 0-7*** eines adressierten Wortes gespeichert.
- Big endian:
  - Least Significant Byte eines Wortes wird an ***Bits 24-31*** eines adressierten Wortes gespeichert.
- Nur relevant, falls Daten als Worte (32 Bit) gespeichert und danach auf kleinere Einheiten zugegriffen wird (Halbworte oder Bytes).
  - Auf welches Byte / Halbwort zugegriffen wird hängt vom Format ab.

Little und Big Endian

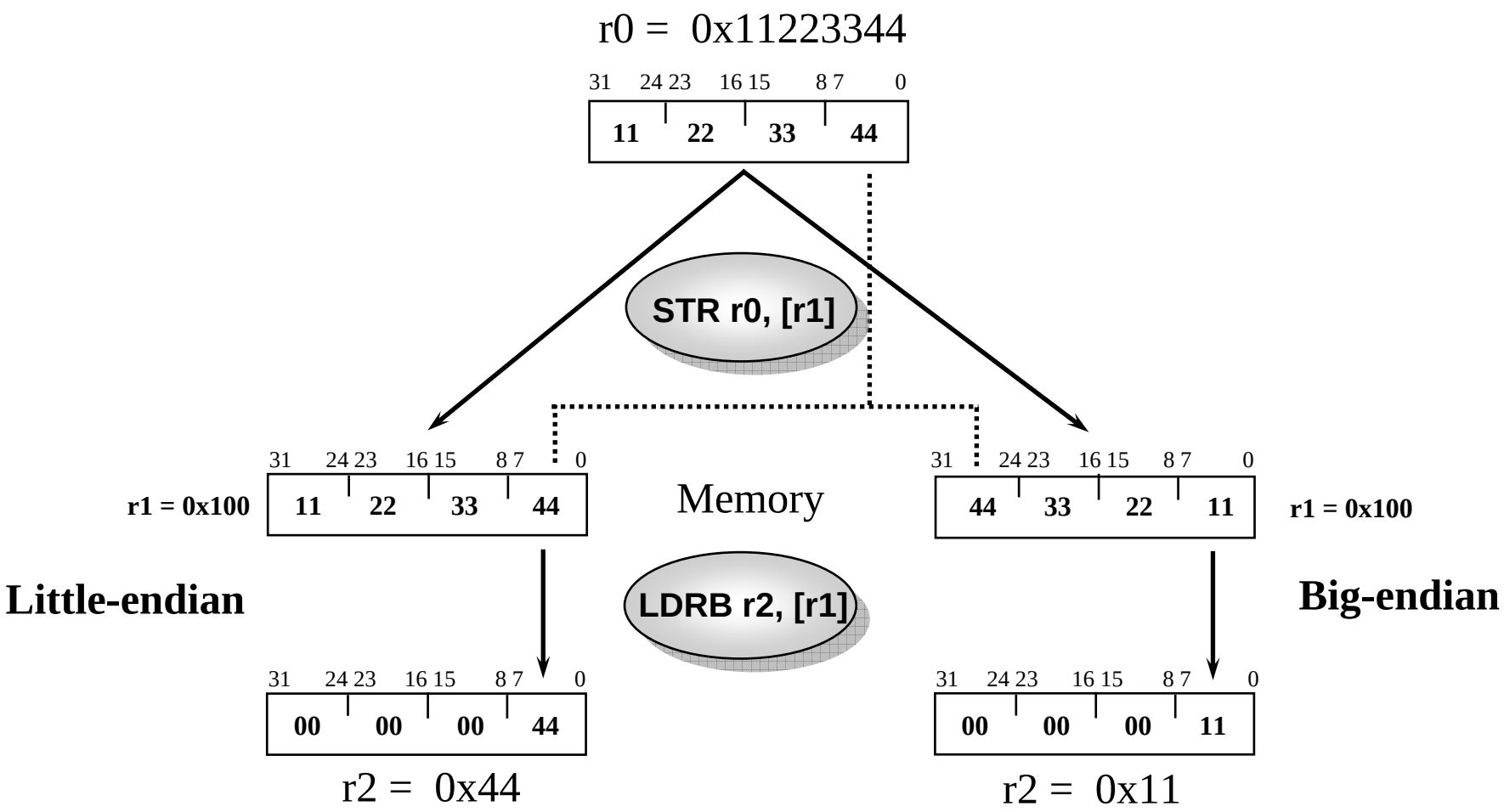


Speicherung von Worten im Big Endian Format



Speicherung von Worten im Little Endian Format

Little und Big Endian



Globale und Lokale Labels

- Werden immer durch ein Symbol (am linken Rand) gefolgt durch einen Doppelpunkt geschrieben
- Beispiel

```
.global main
main:
```

- Labels können durch eine .global Anweisung exportiert werden
- Für die meisten Fälle genügen lokale Labels

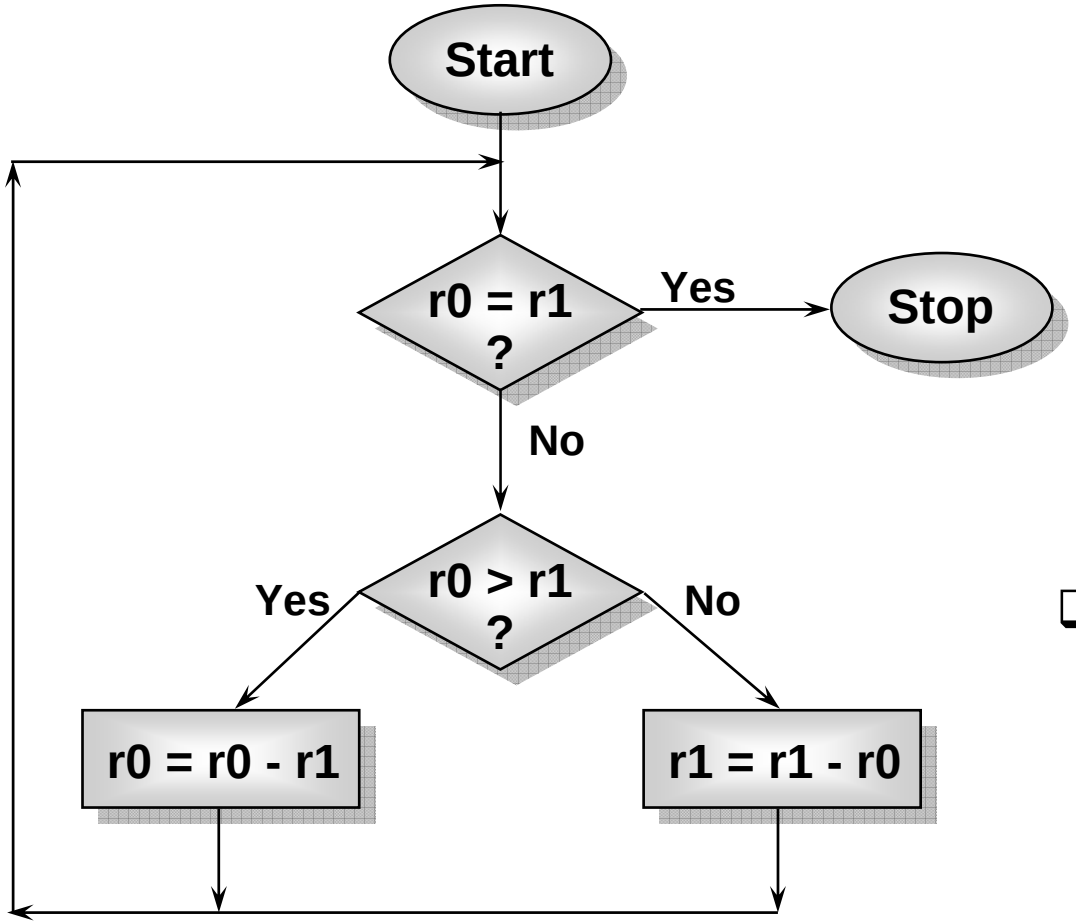
- Im GNU Assembler werden lokale Labels durch eine einstellige Zahl N gekennzeichnet
- die Labels können durch ihre Zahl und die Information f für forward oder b für backward referenziert werden.
- Der Sprung erfolgt jeweils zum nächsten Label mit dieser Nummer in der entsprechenden Richtung

Sprungbefehle

- Sprungbefehle dienen dazu den Kontrollfluss von Programmen zu kontrollieren
- In der Kombination mit Vergleichsbefehlen können mit Ihnen alle wichtigen Kontrollflussanweisungen wie „if.. else“, „while“, „for“ oder „switch“ realisiert werden.
- Das Ziel einer Sprunganweisung ist immer ein Label.

Branch	Interpretation
B	ohne Bedingung
BAL	always
BEQ	equal
BNE	not equal
BPL	plus
BMI	minus
BCC	carry clear
BLO	lower
BCS	carry set
BHS	higher or same
BVC	overflow clear
BVS	overflow set
BGT	greater than
BGE	greater or equal
BLT	less than
BLE	less or equal
BHI	higher
BLS	lower or same

Beispiel GGT



1) “Normaler” Assembler, nur Sprünge können bedingt ausgeführt werden
2) ARM Assembler, alle Instruktionen können bedingt ausgeführt werden (höhere Codedichte)

- Die einzigen Instruktionen, die man benötigt sind CMP, B und SUB.

Beispiel GGT

"Normaler" Assembler

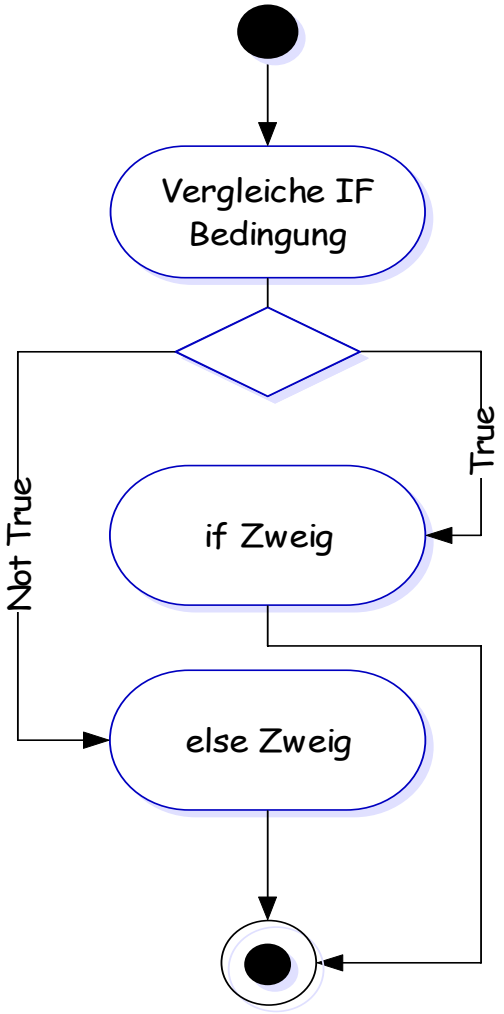
```
ggt     cmp r0, r1      ;Ende erreicht ?????
        beq stop
        blt less        ;falls r0 > r1
        sub r0, r0, r1  ;subtrahiere r1 von r0
        bal ggt
less    sub r1, r1, r0  ;subtrahiere r0 von r1
        bal ggt
stop
```

ARM Conditional Assembler

```
ggt     cmp   r0, r1      ;falls r0 > r1
        subgt r0, r0, r1  ;subtrahiere r1 von r0
        sublt r1, r1, r0  ;andernfalls subtrahiere r0 von r1
        bne   ggt         ;Ende erreicht ?
```

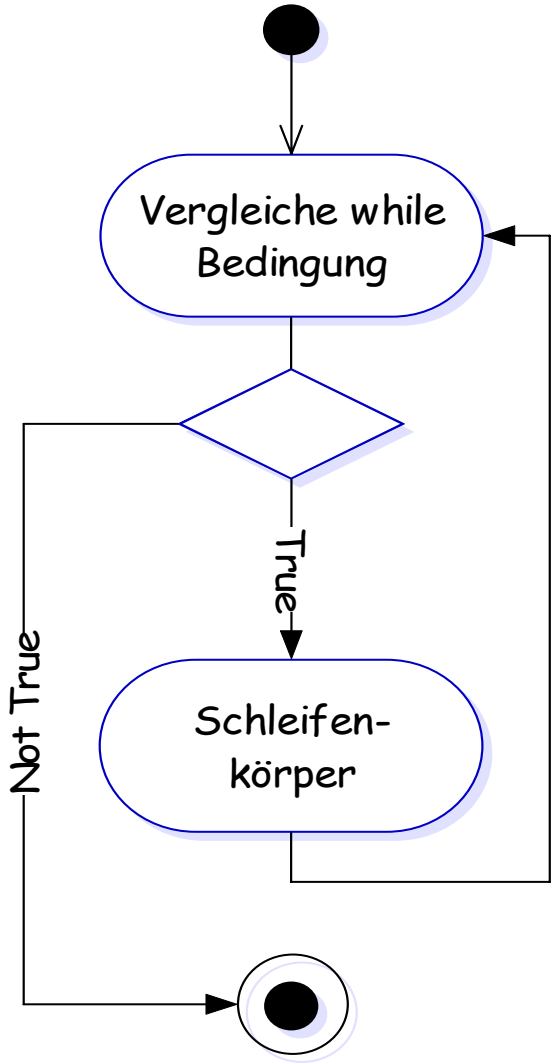
Kontrollfluß: if..else

```
    cmp     ...
    bne     1f
    ...     @ if Zweig
    b   2f
1:
    ...     @ else Zweig
2:
```



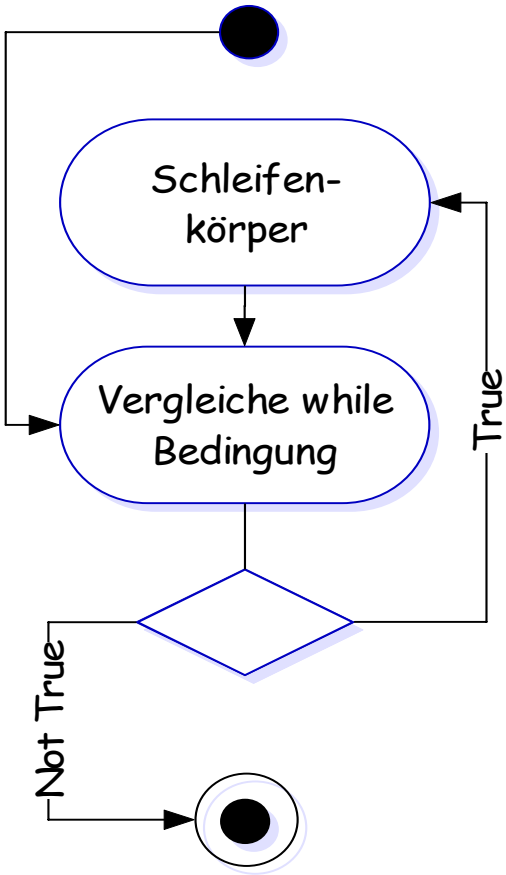
Kontrollfluß: while (1)

```
1:
   cmp ...
   bne      2f
   ...      @ Schleifenkörper
   b 1b
2:
```

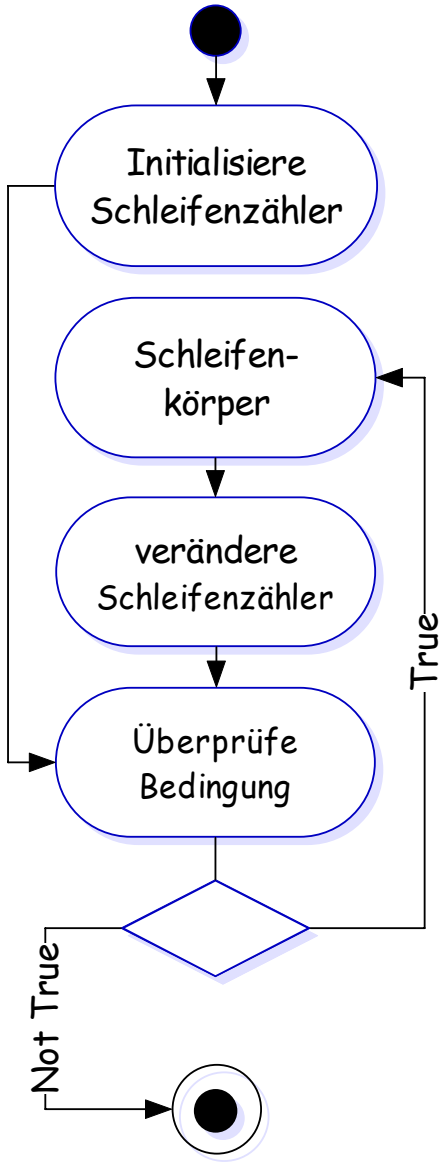


Kontrollfluß: while (2)

```
    b 2f
1:
    ...   @Schleifenkörper
2:
    cmp ...
    beq 1b
```



Zählschleife




```
fill(char *buf, int n) {
  int i;
  for (i = 0 ;i < n; i++  ) {
      buf[i] = 0;
   }
}

fill:
     mov          r3, #0    // i = 0
     cmp          r3, r1
     movge        pc, lr
     mov          r2, r3
.L14:
     strb         r2, [r0, r3]
     add          r3, r3, #1
     cmp          r3, r1
     blt          .L14
     mov          pc, lr
```

For Schleifen

```
void testfor( int a[] )
{
for(int i = 0 ; i < 10; i++)
    a[i] = 0;
}
```

```
testfor:
    mov     r3, #0
    mov     r2, r3
.L6:
    str     r2, [r0, r3, asl #2]
    add     r3, r3, #1
    cmp     r3, #9
    ble     .L6
    mov     pc, lr
```

Das Switch Statement

```
int testswitch( int a, int* b, int c)
{
  switch (a) {
  case 0:
    *b = 0;  break;
  case 1:
    if ( c > 100 ) *b = 0;
    else  *b = 3;
    break;
  case 2:
    *b = 1; break;
  case 3:   break;
  case 4:   *b = 2; break;
  }
}
```

```
testswitch:
    cmp r0, #4
    ldrls pc, [pc, r0, asl #2]
    b     .L2
    .align          2
.L10:
    .word           .L3
    .word           .L4
    .word           .L7
    .word           .L2
    .word           .L9
.L3:
    mov r3, #0
    str    r3, [r1, #0]
    mov pc, lr
```

Das Switch Statement

```
int testswitch( int a, int* b, int c)
{
  switch (a) {
  case 0:
    *b = 0;  break;
  case 1:
    if ( c > 100 ) *b = 0;
    else  *b = 3;
    break;
  case 2:
    *b = 1; break;
  case 3:   break;
  case 4:   *b = 2; break;
  }
}
```

```
.L4:
    cmp     r2, #100
    movgt   r3, #0
    movle   r3, #3
    strgt   r3, [r1, #0]
    strle   r3, [r1, #0]
    mov     pc, lr
.L7:
    mov     r3, #1
    str     r3, [r1, #0]
    mov     pc, lr
.L2:
    mov     pc, lr
.L9:
    mov     r3, #2
    str     r3, [r1, #0]
    mov     pc, lr
```

Unterprogrammtechnik

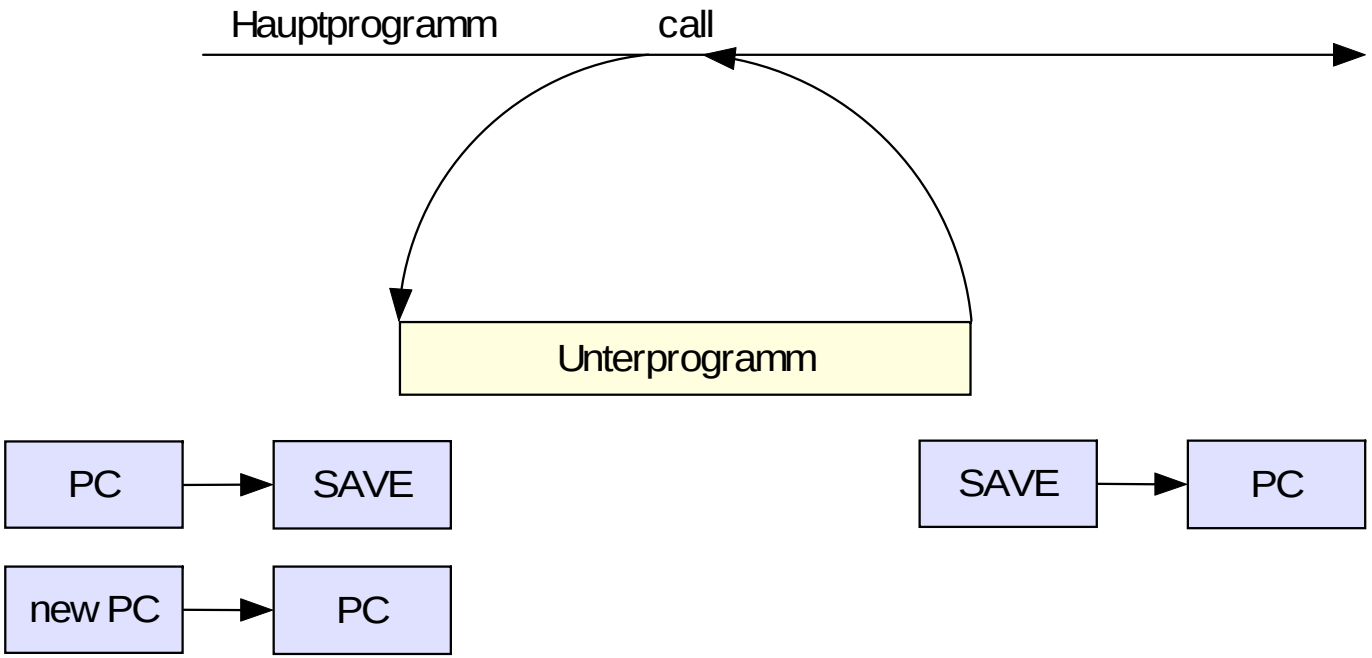
- Der Unterprogramm Aufruf erfolgt durch die Anweisung Branch and Link

  bl Unterprogramm

- Dabei muß Unterprogramm ein zu diesem Zeitpunkt bekanntes Label sein
- Bei der Branch and Link Anweisung wird die nächste Programmadresse hinter dem Aufruf in dem Linkregister (lr) gespeichert und die Unterprogrammadresse in den Program Counter (pc) geladen

- Der Rücksprung aus dem Unterprogramm erfolgt, indem die Rücksprungadresse in den Program Counter geladen wird
- Es wird zwischen Blatt Routinen, die selbst keine Unterprogramme aufrufen (leaf functions), und nicht Blatt Routinen (non leaf functions) unterschieden

Unterprogrammaufruf



APCS Konvention zur Parameterüberg.

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

- Die Parameter 1-4 werden beim Unterprogrammaufruf in den Registern r0-r3 übergeben
- Der Rückgabewert von Funktionen steht im Register r0
- 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 und lr müssen erhalten bleiben

Unterprogrammtechnik 2

- Struktur von „leaf“-Funktionen (wenn nur Register r0-r3 und ip benötigt werden)

```
...
...
mov      pc, lr
```

- Struktur von „non-leaf“ Funktionen

```
stmfd sp!, {regs, lr}
...
...
bl       sub2
...
ldmfd sp!, {regs, pc}
```

Indirekte Unterprogrammaufrufe

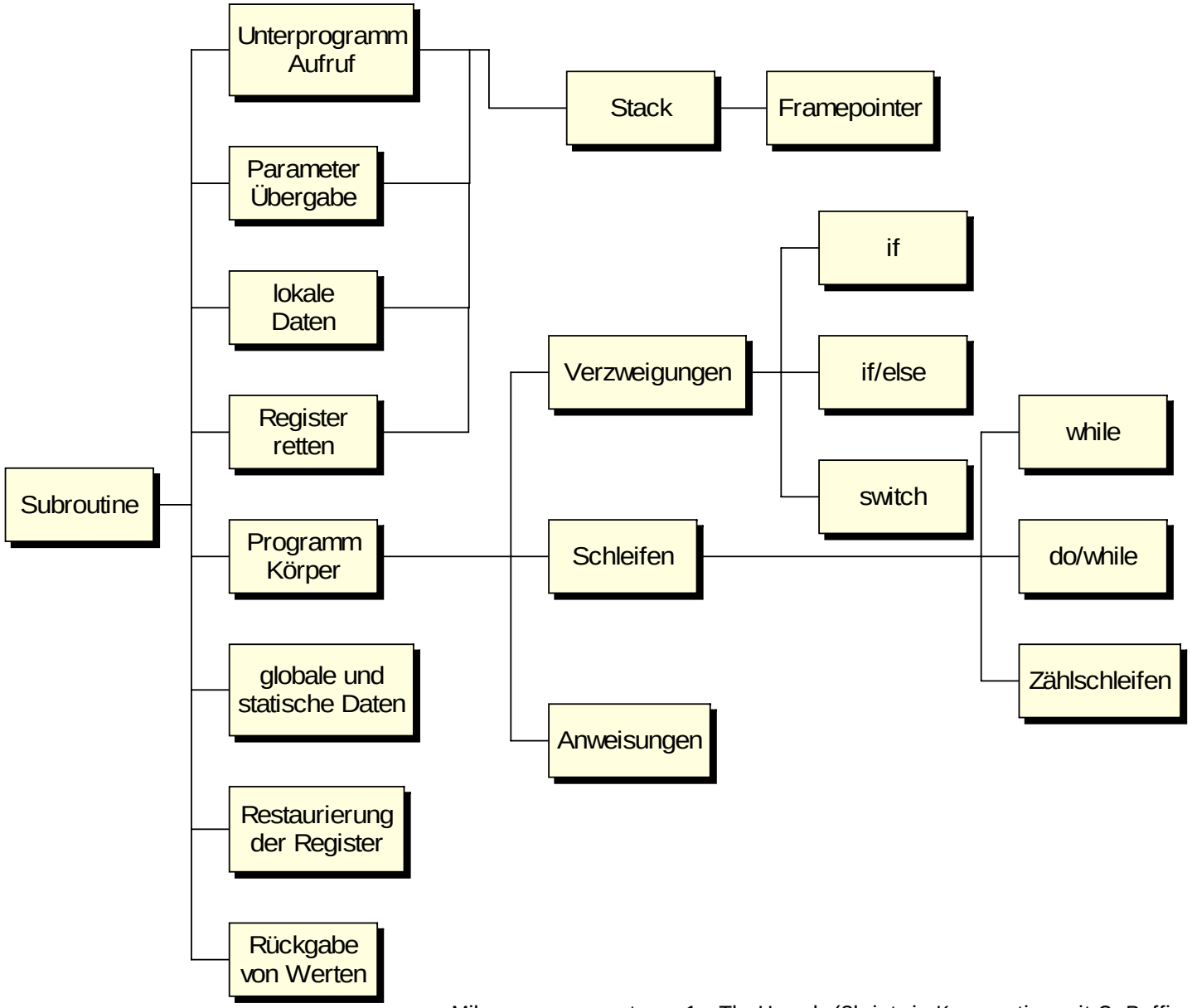
- Will man Unterprogramme aufrufen, deren Adresse zur Compilezeit nicht bekannt ist, muß man den Branch und Link Befehl von Hand nachbilden

- Annahme: aufzurufende Programmadresse steht in r0

```
mov     lr, pc
mov     pc, r0
```

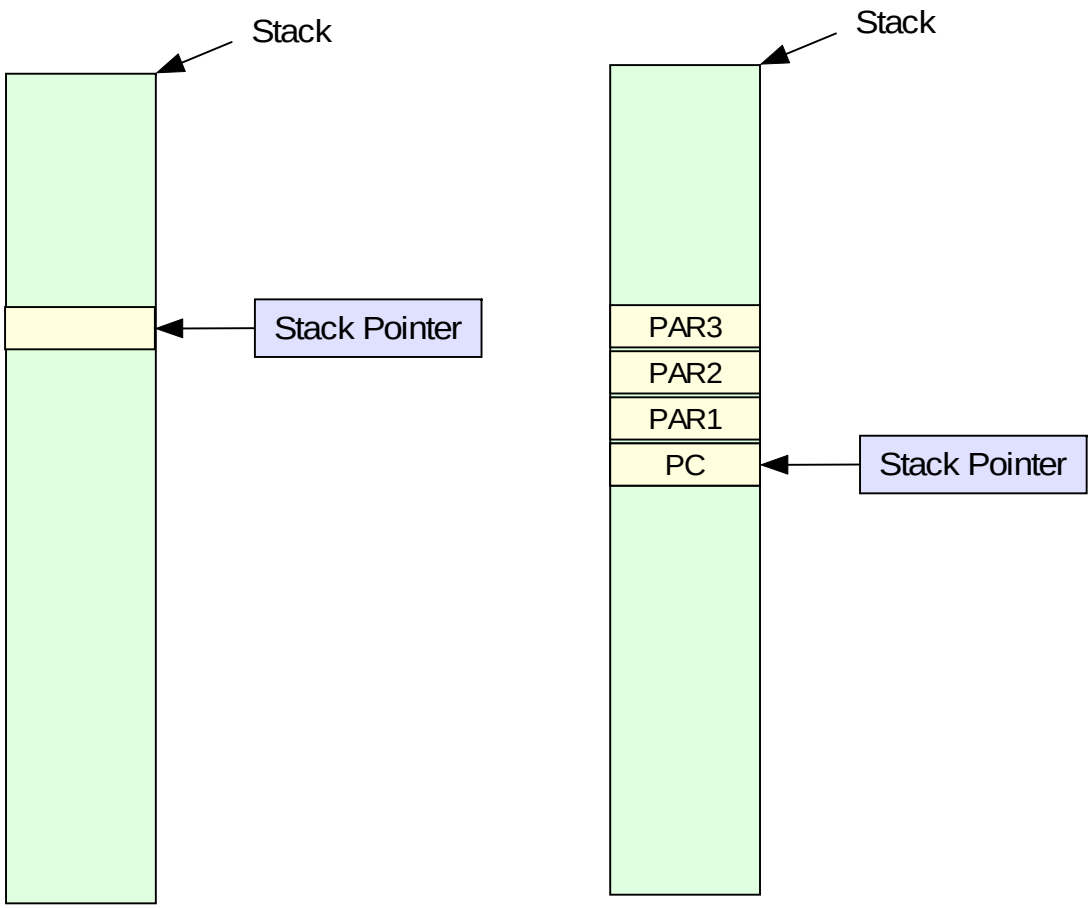
- durch das Pipelining steht im ProgramCounter jeweils die aktuelle Befehlsadresse + 8
- Da jeder Befehl eine Länge von 4 Byte besitzt zeigt das Linkregister nach dieser Befehlsfolge genau auf die Rücksprungadresse

Konzept eines Unterprogramms

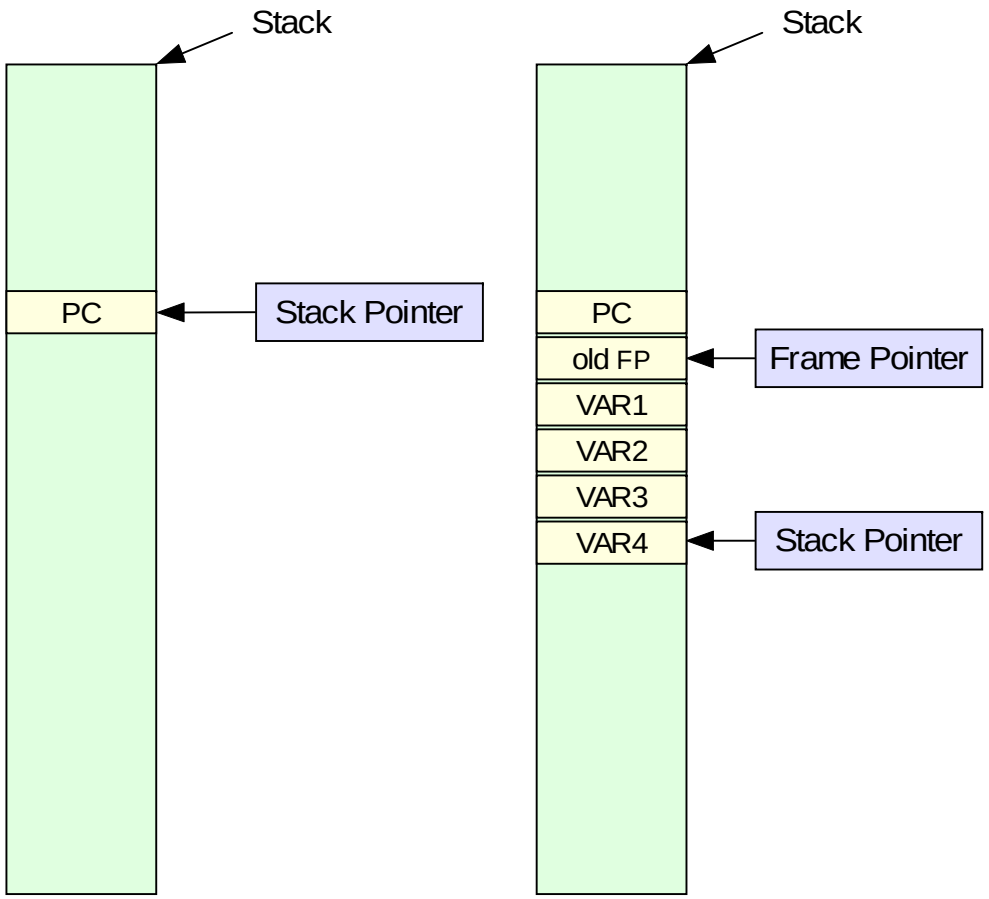


Unterprogramm mit Parameterübergabe

CISC



Lokale Parameter (CISC)



Lokale Parameter (Risc)

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/Scratch
PC	PC

Die Verwendung der ARM Register bei Unterprogrammaufrufen

Ablauf eines Unterprogramms

1. Framepointer, Register retten
2. SP -> FP
3. Platz reservieren
4. **Programmausführung**
5. FP -> SP
6. Framepointer, Register restaurieren

- Eingangssequenz

```
stmfd sp!, {fp, lr}
mov       fp, sp
sub sp, sp, #12     @ 3 Worte
...
ldr r1, [fp, #-4]
...
```

- Ausgangssequenz

```
mov sp, fp
ldmfd sp!, {fp, pc}
```

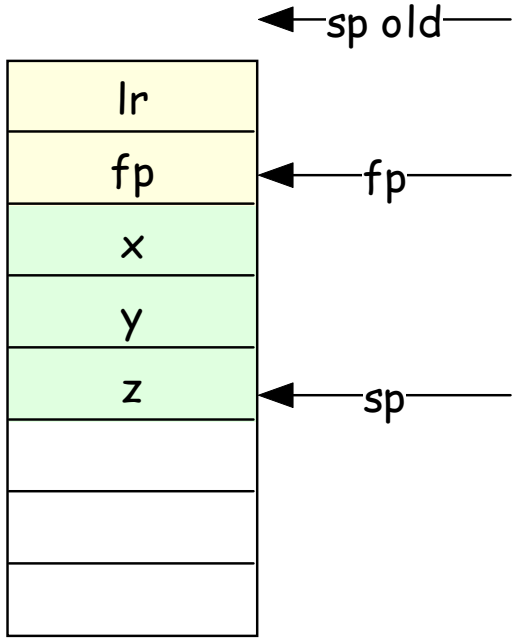
Stackframes

- Eingangssequenz

```
stmfd    sp!, {fp, lr}
mov      fp, sp
sub      sp, sp, #12   @ 3 Worte
...
ldr      r1, [fp, #-4]
...
```

- Ausgangssequenz

```
mov      sp, fp
ldmfd    sp!, {fp, pc}
```



Ein einfaches Assembler Programm

```
//program1.c
#include <stdio.h>

int x = 2;
int y = 3;
int z;

int main( void)
{
  z = x + y;
  printf("z = %d\n", z);
  return (1);
}
```

```
    .data
    .align      2
    .global     x
x:
    .word       2
    .global     y
y:
    .word       3
```

Ein einfaches Assembler Programm

```
    .text
    .align      2
    .global     main
main:
    stmfd       sp!, { lr }
    ldr         r2, .L2
    ldr         r3, .L2+4
    ldr         ip, [r3, #0]
    ldr         r3, [r2, #0]
    add         ip, ip, r3
    ldr         r3, .L2+8
    mov         r1, ip
    str         ip, [r3, #0]
    ldr         r0, .L2+12
    bl          printf
    mov         r0, #1
    ldmfd       sp!, {pc}
```

```
.L3:
    .align      2
.L2:
    .word       y
    .word       x
    .word       z
    .word       .LC0
    .text
    .align      2
.LC0:
    .ascii"z = %d\n\000“

    .comm       z, 4

    .end
```

Unterprogramm Beispiel my_max

```
// my_max.c
#include <stdlib.h>

void my_max(int a, int b, int* c)
{
    if (a > b)
        *c = a;
    else
        *c = b;
}
```

Assembler Programm ohne Optimierung

```
// my_max.S
    .text
    .align          2
    .global         my_max
    .type my_max,function
my_max:
    mov ip, sp
    stmfd           sp!, {fp, ip, lr, pc}
    sub  fp, ip, #4
    sub  sp, sp, #12
    str    r0, [fp, #-16]
    str    r1, [fp, #-20]
    str    r2, [fp, #-24]
    ldr    r2, [fp, #-16]
    ldr    r3, [fp, #-20]
    cmp r2, r3
    ble    .L2
    ldr    r2, [fp, #-16]
    ldr    r3, [fp, #-24]
    str    r2, [r3, #0]
    b      .L3
.L2:
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    str    r2, [r3, #0]
.L3:
    ldmea           fp, {fp, sp, pc}
.Lfe1:
    .size my_max,.Lfe1-my_max
```

Unterprogramm mit Optimierung

```
#include <stdlib.h>


void my_max(int a, int b, int* c)
{
    if (a > b)
        *c = a;
    else
        *c = b;
}
```

```
    .text
    .align      2
    .global     my_max
    .type       my_max,function
my_max:
    cmp         r0, r1
    strge       r0, [r2, #0]
    strlt       r1, [r2, #0]
    mov         pc, lr
.Lfe1:
    .size       my_max,.Lfe1-my_max
```

APCS Konvention für Strukturen

Strukturen werden zurückgegeben durch einen Zeiger auf den Speicher, wo die Struktur abgelegt ist.

Funktionen, die eine Struktur zurückgeben, z.B.

```
struct s fct(int i)
```

wird übersetzt als

```
void fct(struct s *result, int x)
```

APCS Konvention für Strukturen

Beispiel C:

```
typedef struct two_ch_struct
{ char ch1;
  char ch2;
} two_ch;

two_ch max( two_ch a, two_ch b )
{
  return (a.ch1>b.ch1) ? a : b;
}
int main(void) {   // Hauptprogramm
two_ch a,b,c;
a.ch1 = a.ch2 = b.ch2 = 'A';
b.ch1 = 'B';
c = max( a, b );
}
```

Beispiel ARM Assembler:

```
max
        MOV     ip,sp
        STMDB sp!,{a1-a3,fp,ip,lr,pc}
        SUB     fp,ip,#4
        LDRB    a3,[fp,#-&14]
        LDRB    a2,[fp,#-&10]
        CMP     a3,a2
        SUBLE   a2,fp,#&10
        SUBGT   a2,fp,#&14
        LDR     a2,[a2,#0]
        STR     a2,[a1,#0]
        LDMDB   fp,{fp,sp,pc}
```

Optimierung ?

Strcmp / while Schleifen

```
int strcmp(const char* str1, const
    char* str2)
{
    while( *str1 && *str1 == *str2 )
    {
        str1++; str2++;
    }
    return ( *str1 - *str2 );
}
```

```
strcmp:
    ldrb        r2, [r0, #0]
    cmp         r2, #0
    beq         .L3
    ldrb        r3, [r1, #0]
    cmp         r2, r3
    beq         .L6
```

```
.L3:
    ldrb        r0, [r0, #0]
    ldrb        r3, [r1, #0]
    rsb         r0, r3, r0
    mov         pc, lr
.L6:
    ldrb        r3, [r0, #1]!
    and         r2, r3, #255
    cmp         r2, #0
    add         r1, r1, #1
    beq         .L3
    ldrb        r3, [r1, #0]
    cmp         r2, r3
    bne         .L3
    b           .L6
```

strcmp

```
int strcmp(const char *s1, const
    char *s2)
{
  unsigned int  u1, u2;

  while (1)
    {
      u1 = (unsigned int) *s1++;
      u2 = (unsigned int) *s2++;
      if (u1 != u2)
            return u1 - u2;
      if (u1 == '\0')
            return 0;
    }
}
```

```
strcmp:
    mov         r2, r0
.L7:
    ldrb        r0, [r2], #1
    ldrb        r3, [r1], #1
    cmp         r0, r3
    rsbne       r0, r3, r0
    movne       pc, lr
    cmp         r0, #0
    bne         .L7
    mov         pc, lr
```

lenstr

```
int lenstr( char *pt )
{
    int len = 0;
    while ( *pt++ ) len++;
    return len;
}
```

```
lenstr:
    ldrb    r3, [r0], #1
    cmp     r3, #0
    mov     r2, #0
    beq     .L6
.L5:
    ldrb    r3, [r0], #1
    cmp     r3, #0
    add     r2, r2, #1
    bne     .L5
.L6:
    mov     r0, r2
    mov     pc, lr
```

ggt

```
int ggt(int a, int b)
{
    int c = a%b;
    if ( c == 0 ) return b;
    else return ggt( b, c );
}
```

```
ggt:
    mov     ip, sp
    stmfd   sp!, {r4, fp, ip, lr, pc}
    mov     r3, r0
    sub     fp, ip, #4
    mov     r4, r1
.L4:
    mov     r0, r3
    mov     r1, r4
    bl      __modsi3
    cmp     r0, #0
    mov     r3, r4
    moveq   r0, r4
    ldmeqea fp, {r4, fp, sp, pc}
    mov     r4, r0
    b       .L4
```

strcpy

```
void strcpy(char *str1, char *str2)
{
    while (*str1++ = *str2++);
}
```

```
strcpy:
.L9:
ldrb        r3, [r1], #1
cmp     r3, #0
strb        r3, [r0], #1
moveq   pc, lr
b       .L9
```

strchr

```
char * strchr (const char
   *s, int c)
{
  char *rtnval = 0;

  do {
    if (*s == c)
       rtnval = (char*) s;
  } while (*s++);
  return (rtnval);
}
```

```
strchr:
   mov      r2, #0
.L2:
   ldrb     r3, [r0, #0]
   cmp      r3, r1
   moveq    r2, r0
   cmp      r3, #0
   add      r0, r0, #1
   bne      .L2
   mov      r0, r2
   mov      pc, lr
```

strcat

```
void strcat( char* str1,
   const char *str2 )
{
    while( *str1 ) {
   str1++;
    }
    while( *str1++ =
   *str2++ );
}
```

```
strcat:
    ldrb        r3, [r0, #0]
    cmp         r3, #0
    beq         .L6
.L5:
    ldrb        r3, [r0, #1]!
    cmp         r3, #0
    bne         .L5
.L6:
    ldrb        r3, [r1], #1
    tst         r3, #255
    strb        r3, [r0], #1
    bne         .L6
    mov         pc, lr
```

strtol

```
// strtol.c
long my_strtol( const char *nptr,
    char **endptr, int base )
{
    const char *s = nptr;
    unsigned long acc;
    int c;
    int neg = 0;

    do {
        c = *s++;
    } while (isspace(c));
```

```
// strtol.S
my_strtol:
    mov     ip, sp
    stmfd   sp!, {r4, r5, r6, r7, r8,
                  fp, ip, lr, pc}
    sub     fp, ip, #4
    mov     r6, r2
    mov     r5, r0
    mov     r8, #0
.L49:
    ldrb    r4, [r5], #1
    mov     r0, r4
    bl      isspace
    cmp     r0, #0
    bne     .L49
```

strtol - Teil 2

```
if (c == '-') {
        neg = 1;
        c = *s++;
    } else if (c == '+')
        c = *s++;

    if ((base == 0 || base == 16)
     &&       c == '0' && (*s == 'x'
     || *s == 'X')) {
        c = s[1];
        s += 2;
        base = 16;
    }

    if (base == 0)
        base = c == '0' ? 8 : 10;
```

```
    cmp         r4, #45
    ldreqb      r4, [r5], #1
    moveq       r8, #1
    beq         .L51
    cmp         r4, #43
    ldreqb      r4, [r5], #1
.L51:
    cmp         r6, #16
    cmpne       r6, #0
    bne         .L53
    cmp         r4, #48  @ c ==  '0'
    beq  .L69
.L53:
    cmp         r6, #0    @ base ==0
    beq         .L70
```

strtol – Teil 3

```
for (acc = 0;; c = *s++) {
        if (isdigit(c))
            c -= '0';
        else if (isalpha(c))
            c -= isupper(c) ? 'A' -
    10 : 'a' - 10;
        else
            break;
        if (c >= base)
            break;
    acc *= base;
    acc += c;
    }
```

```
.L54:
    mov     r7, #0      @ acc = 0
.L57:
    mov     r0, r4      @Parameter c
    bl      isdigit
    mov     r3, r0
    cmp     r3, #0
    mov     r0, r4      @Parameter c
    subne   r4, r4, #48 @ c -= '0';
    bne     .L61
    bl      isalpha
    mov     r3, r0
    cmp     r3, #0
    mov     r0, r4
    beq     .L58
    bl      isupper
    cmp     r0, #0
    sub     r3, r4, #55
    subeq   r3, r4, #87
    mov     r4, r3
```

strtol – Teil 4

```
if (neg)
        acc = -acc;
    return acc;
}
```

```
.L61:
        cmp     r4, r6          @ c >= base
        mlalt   r7, r6, r7, r4
        ldrltb  r4, [r5], #1    @ zero_extendqisi2
        blt     .L57
.L58:
        cmp     r8, #0
        rsbne   r7, r7, #0
        mov     r0, r7
        ldmea   fp, {r4, r5, r6, r7, r8, fp, sp, pc}
.L70:
        cmp     r4, #48         @ c == '0'
        movne   r6, #10         @ base = 10
        moveq   r6, #8          @ base = 8
        b       .L54
.L69:
        ldrb    r3, [r5, #0]    @ zero_extendqisi2
        cmp     r3, #88
        cmpne   r3, #120
        ldreqb  r4, [r5, #1]    @ zero_extendqisi2
        moveq   r6, #16
        addeq   r5, r5, #2
        b       .L53
```

Beispiel C Programm

```
// reverse.c                          <- Statische Daten
char buf[] = "Hello world";
                                      <- Parameter Übergabe
void reverse(char* buf)
{                                     <- Lokale Parameter
  char* start;
  char* end;
  char temp;                          <- Anweisungen

  start = buf;
  end = buf;
                                      <- Schleife
  while( *end ) {
    end++;
  }
  end -= 1;
    while ( start < end ) {
    temp = *start;
    *start++ = *end;
    *end-- = temp;
  }
}
```

```
void main()
{
        reverse(buf);
        printf (“ %s \n“, buf);
}
```

Assemblerprogramm - ARM

```
// reverse.S
.gcc2_compiled.:
            .global     buf
.data
            .align      2
            .type        buf,object
            .size        buf,12
buf:
            .ascii      "Hello world\000"
.text
            .align      2
            .global     reverse
            .type        reverse,function
reverse:
            mov         ip, sp
            stmfd       sp!, {fp, ip, lr, pc}
            sub         fp, ip, #4
            ldrb        r3, [r0, #0]

            mov         r1, r0
            cmp         r3, #0
            beq         .L4
.L5:
            ldrb        r3, [r1, #1]!
            cmp         r3, #0
            bne         .L5
.L4:
            sub         r1, r1, #1
            cmp         r0, r1
            ldmcsea     fp, {fp, sp, pc}
.L9:
            ldrb        r3, [r1, #0]
            ldrb        r2, [r0, #0]
            strb        r3, [r0], #1
            strb        r2, [r1], #-1
            cmp         r0, r1
            bcc         .L9
            ldmea       fp, {fp, sp, pc}
```

Assemblerprogramm - ARM

```
            .global     buf
.data
            .align      2
            .type        buf,object
            .size        buf,12
buf:
            .ascii      "Hello world\000"
.text
            .align      2
            .global     reverse
            .type        reverse,function
reverse:

            ldrb        r3, [r0, #0]
            mov         r1, r0
            cmp         r3, #0
            beq         .L4
.L5:
            ldrb        r3, [r1, #1]!
            cmp         r3, #0
            bne         .L5
```

```
.L4:
            sub         r1, r1, #1
            cmp         r0, r1
            movcs       pc, lr
.L9:
            ldrb        r3, [r1, #0]
            ldrb        r2, [r0, #0]
            strb        r3, [r0], #1
            strb        r2, [r1], #-1
            cmp         r0, r1
            bcc         .L9
            mov         pc, lr
```

Beispiel atoi

```
int atoi( char * str )
{
int sum = 0;
int c;

while ( (c = * str ++) != 0 ) {
        sum = sum * 10 + ( c - '0');
}
return sum ;
}

int main(void) {
        return( atoi("123") );
}
```

```
atoi
        MOV     a2,#0
        LDRB    a3,[a1],#1
        TEQ     a3,#0
        BEQ     J5
J4
        ADD     a2,a2,a2,LSL #2
        ADD     a2,a3,a2,LSL #1
        SUB     a2,a2,#&30
        LDRB    a3,[a1],#1
        TEQ     a3,#0
        BNE     J4
J5
        MOV     a1,a2
        MOV     pc,lr
```

Beispiel atoi_reverse

```
int atoi_rev(char *str)
{
int c;
if ((c = *str++) == 0)
  return 0;
else
   return(atoi_rev(str)*10 + (c - '0') );
}

int main(void) {
return( atoi_rev("123") );
}
```

```
atoi_rev

        MOV     ip,sp
        STMDB sp!,{v1,fp,ip,lr,pc}
        SUB     fp,ip,#4
        LDRB   v1,[a1],#1
        TEQ     v1,#0
        MOVEQ    a1,#0
        LDMEQDB  fp,{v1,fp,sp,pc}
        BL       atoi_rev
        ADD      a1,a1,a1,LSL #2
        ADD      a1,v1,a1,LSL #1
        SUB      a1,a1,#&30
        LDMDB fp,{v1,fp,sp,pc}
```

64 Bit Addition

64-bit Addition:

```
// Datei add64_1.c
typedef struct int64_struct {
unsigned int lo;
unsigned int hi;
} int64;

void add_64(int64 *dest, int64 *src1, int64 *src2) {
unsigned hibit1=src1->lo >> 31, hibit2=src2->lo >> 31, hibit3;
  dest->lo=src1->lo + src2->lo;
  hibit3=dest->lo >> 31;
  dest->hi=src1->hi + src2->hi +
          ((hibit1 & hibit2) || (hibit1!= hibit3));
  return;
}
```

64 Bit Addition

```
add_64
        STMDB   sp!,{v1,lr}
        LDR     v1,[a2,#0]
        MOV     a4,v1,LSR #31
        LDR     ip,[a3,#0]
        MOV     lr,ip,LSR #31
        ADD     ip,v1,ip
        STR     ip,[a1,#0]
        MOV     ip,ip,LSR #31
        LDR     a2,[a2,#4]
        LDR     a3,[a3,#4]
        ADD     a2,a2,a3
        TST     a4,lr
        TEQEQ   a4,ip
        MOVNE   a3,#1
        MOVEQ   a3,#0
        ADD     a2,a2,a3
        STR     a2,[a1,#4]!
        LDMIA   sp!,{v1,pc}
```

((hibit1 & hibit2) || (hibit1!= hibit3))

64 Bit Addition

Erzeugtes Assembler Programm ohne Betrachtung der hibit‘s

```
add_64neu
        LDR     a4,[a2,#0]
        LDR     ip,[a3,#0]
        ADDS    a4,a4,ip
        STR     a4,[a1,#0]
        LDR     a2,[a2,#4]
        LDR     a3,[a3,#4]
        ADC     a2,a2,a3
        STR     a2,[a1,#4]!
        MOV     a1,#1
        MOV     pc,lr
```

In ARM Assembler Zugriff auf Statusbits möglich

daher effizienteres Programm

64 Bit Produkt von zwei 32 Bit Integer

```
// Beginn: a1 und a2 beinhalten die beiden Faktoren als 32-Bit Integer
// Ende:   a1 und a2 beinhalten das Ergebnis (a1 Bits 0-31, a2 Bits 32-63)
mul64
    MOV     ip, a1, LSR #16         ; ip = a_hi
    MOV     a4, a2, LSR #16         ; a4 = b_hi
    BIC     a1, a1, ip, LSL #16     ; a1 = a_lo
    BIC     a2, a2, a4, LSL #16     ; a2 = b_lo
    MUL     a3, a1, a2              ; a3 = a_lo * b_lo      (m_lo)
    MUL     a2, ip, a2              ; a2 = a_hi * b_lo      (m_mid1)
    MUL     a1, a4, a1              ; a1 = a_lo * b_hi      (m_mid2)
    MUL     a4, ip, a4              ; a4 = a_hi * b_hi      (m_hi)
    ADDS    ip, a2, a1              ; ip = m_mid1 + m_mid2   (m_mid)
    ADDCS   a4, a4, #&10000         ; a4 = m_hi + carry      (m_hi')
    ADDS    a1, a3, ip, LSL #16     ; a1 = m_lo + (m_mid<<16)
    ADC     a2, a4, ip, LSR #16     ; a2 = m_hi' + (m_mid>>16) + carry
    MOV     pc, lr
```

Festkomma-Arithmetik

Wie rechnet man

```
float          Y, M,X,B;
Y = M * X + B;
```

auf Prozessoren ohne Floating Point Unterstützung ?

Festkomma-Arithmetik:

Festlegung des Dezimalpunktes, z.B. Bit 24

Festkommazahl Format 8.24

Minimalwert: -2^(32-24-1) = -128,000 000 00

Maximalwert: 2^(32-24-1) – 1/2^24 = 127,999 999 94

Auflösung: 1/2^24 = 0,000 000 06

Festlegung des Dezimalpunktes ist Kompromiß zwischen Auflösung und Zahlenbereich

Festkomma-Arithmetik

Lösung in C:

int32 Y,M,X,B; Zahlen im Festkommaformat 8.24

// Y = M * X + B

Y = ((int64) M * (int64) X + (int64) B << 24) >> 24;

Addition:

A(8.24) + B(8.24) = C(8.24)

Multiplikation:

A(8.24) * B(8.24) = C(16.48)   // 64 Bit Produkt

C muss um 24 Bits nach rechts geschoben werden

->> C(8.24)

Multiplikation + Addition:

siehe C-Beispiel oben für Y = M*X + B

Festkomma-Arithmetik

Beispiel für Multiplikation + Addition:

Y = M * X + B = 1.5 * 2.0 + 1.0 = 4.0

Festkommaformat 28.4:

M = 0x0000018 entspricht dezimal 1.5
X = 0x0000020 entspricht dezimal 2.0
B = 0x0000010 entspricht dezimal 1.0

Ergebnis:
Y = 0x0000040 entspricht dezimal 4.0

Division durch eine Konstante (1)

Teiler ist eine Zweierpotenz

```
MOV     a2, a1, lsr #5      ; Division durch 32 (unsigned)
MOV     a2, a1, asr #10     ; Division durch 1024 (signed)
```

ansonsten: Mulitplikation mit Kehrwert

y ist Konstante

x/y = x*(1/y) 1/y ist 0.32 Festkommazahl

= x *(2^32/y))/2^32 2^32/y ist 32.0 Festkommazahl

= x *(2^32/y)) >> 32 gibt das Highwort eines 64 Bit Produkts von x und 2^32/y zurück

Division durch eine Konstante (2)

y	(2^32/y)
2	10000000000000000000000000000000
3	01010101010101010101010101010101
4	01000000000000000000000000000000
5	00110011001100110011001100110011
6	00101010101010101010101010101010
7	00100100100100100100100100100100
8	00100000000000000000000000000000
9	00011100011100011100011100011100
10	00011001100110011001100110011001
11	00010111010001011101000101110100
12	00010101010101010101010101010101
13	00010011101100010011101100010011
14	00010010010010010010010010010010
15	00010001000100010001000100010001
16	00010000000000000000000000000000

Division durch eine Konstante (3)

Für Spezialfälle: Ausnutzen des „Wiederholungsmusters“

Methode analog zur Multiplikation einer Konstante

Modifikation der Methode, indem nach rechts verschoben wird

Effizienz durch Berechnung nur des Highworts

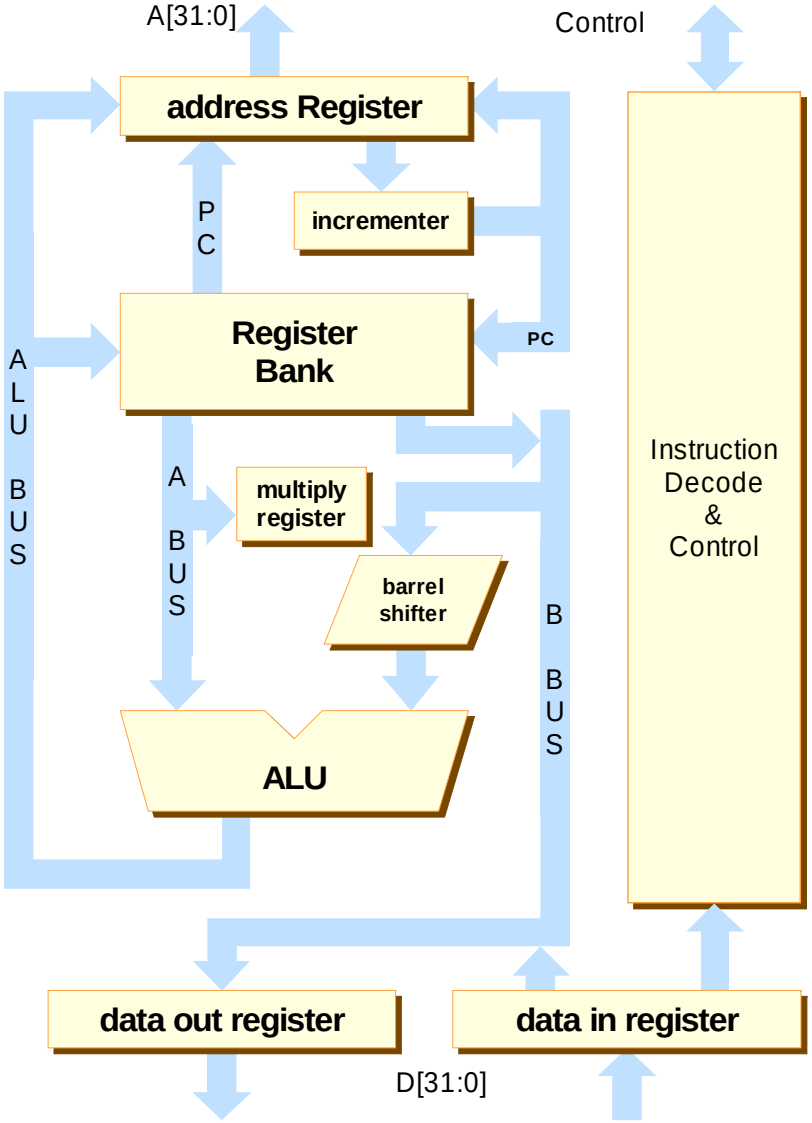
Beispiel: Division durch 10 (x ist der Dividend):

```
SUB     a1, x, x, lsr #2      ; a1 = x*0.11000000000000000000000000000000
ADD     a1, a1, a1, lsr #4    ; a1 = x*0.11001100000000000000000000000000
ADD     a1, a1, a1, lsr #8    ; a1 = x*0.11001100110011000000000000000000
ADD     a1, a1, a1, lsr #16   ; a1 = x*0.11001100110011001100110011001100
MOV     a1, a1, lsr #3        ; a1 = x*0.00011001100110011001100110011001
```

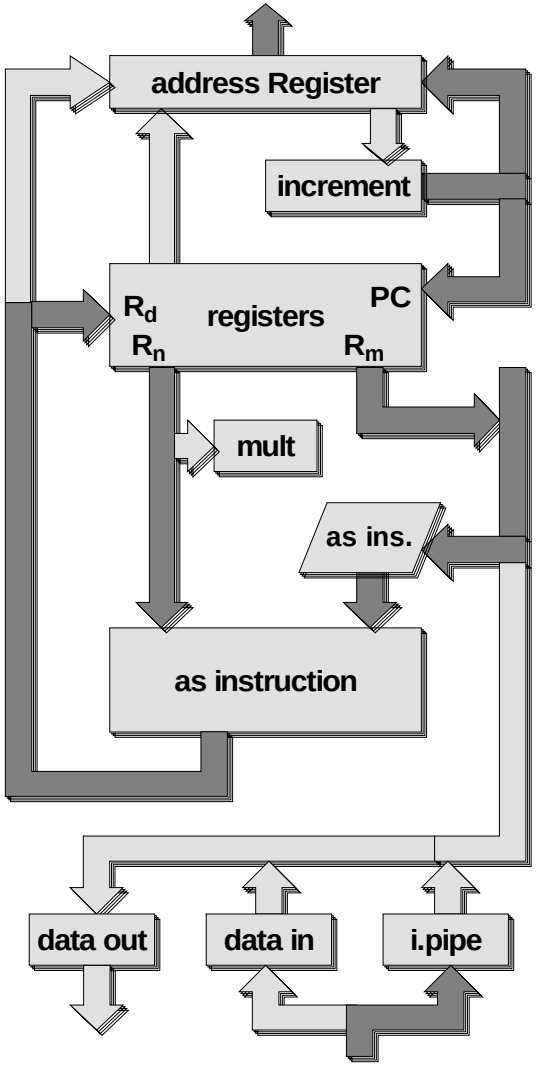
SUB berechntet: a1 = x - x/4 = x - x*0.01 = x*0.11

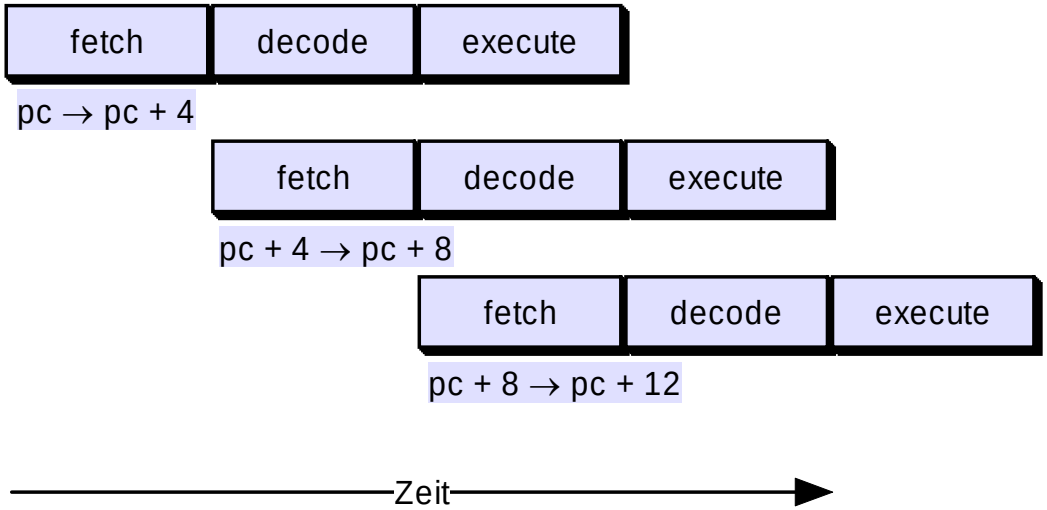
(alle Zahlen im Binärsystem)

ARM7TDMI Struktur



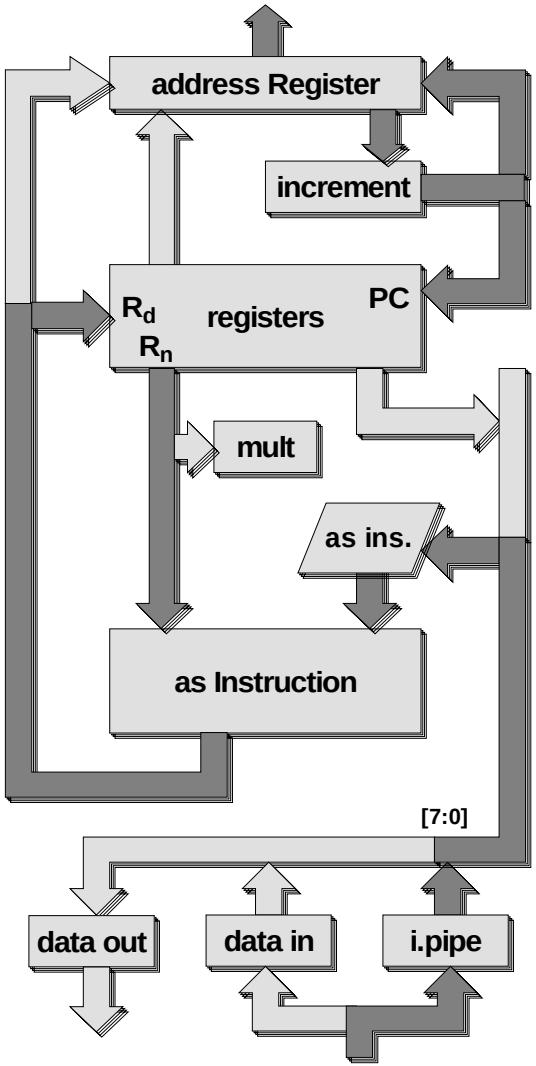
Register-Register Operation





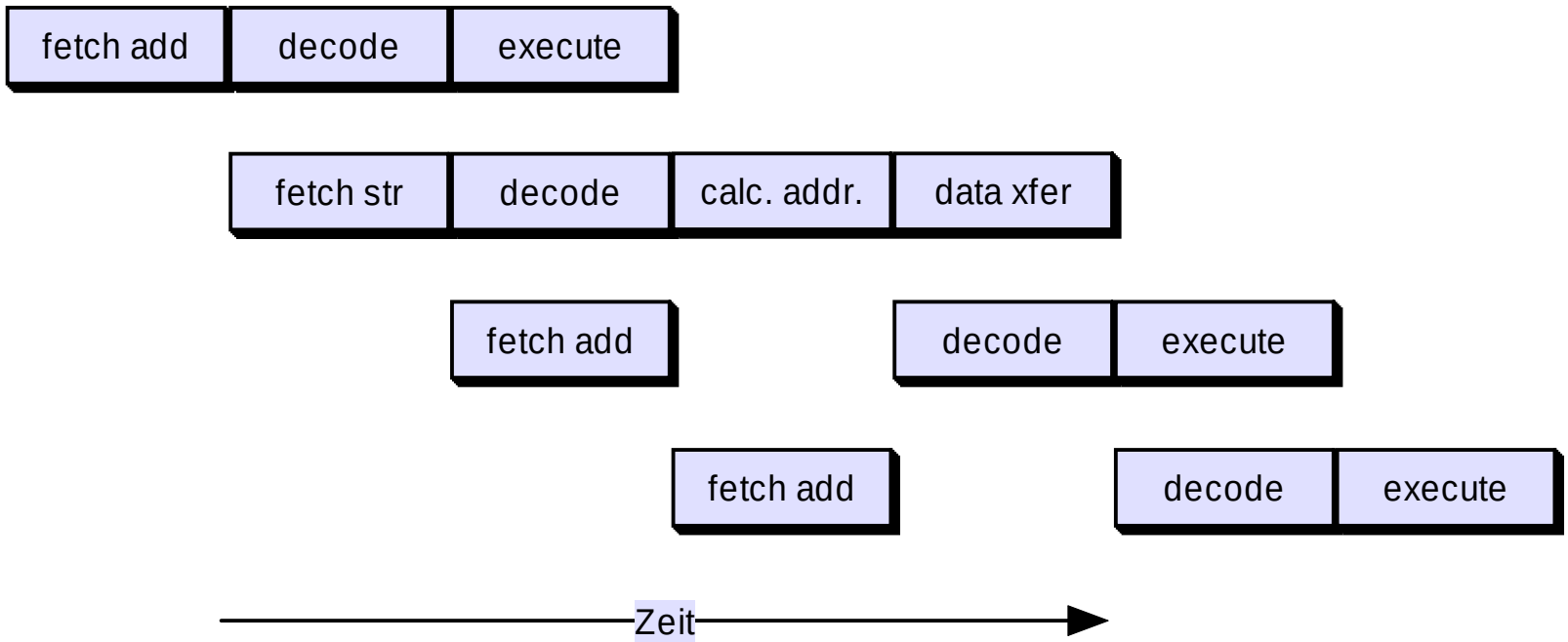
Register - Register Operation

Register-immediate Operation

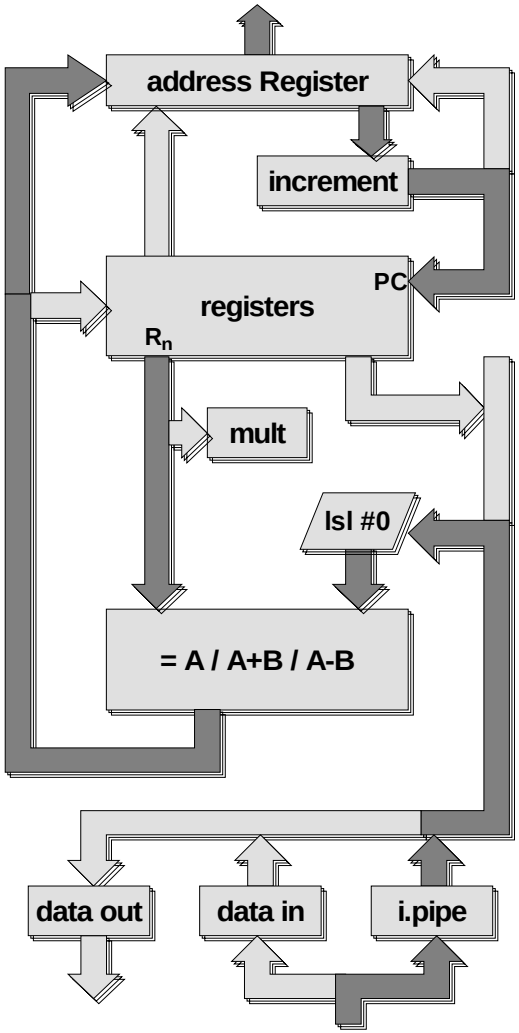


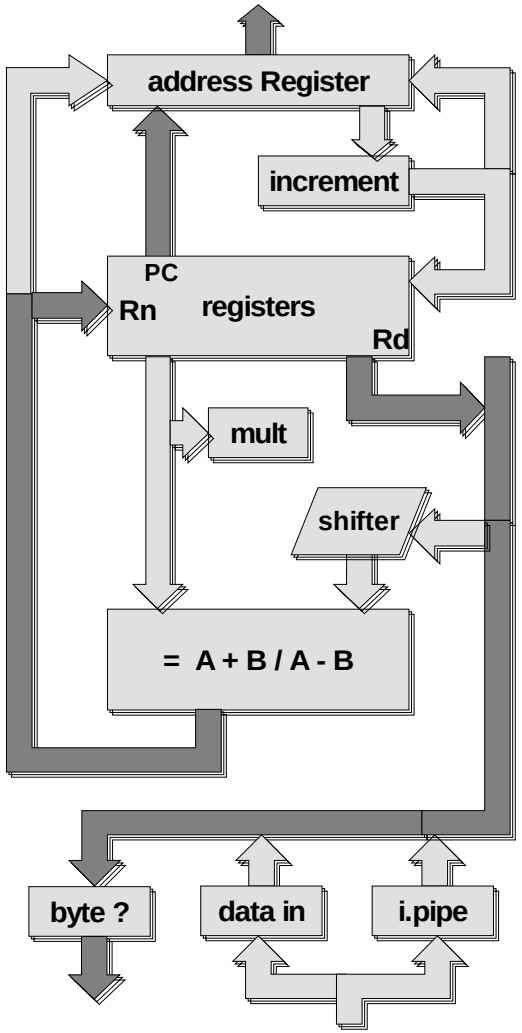
Register - immediate Operation

Daten Speichern



Store Register Datenpfad

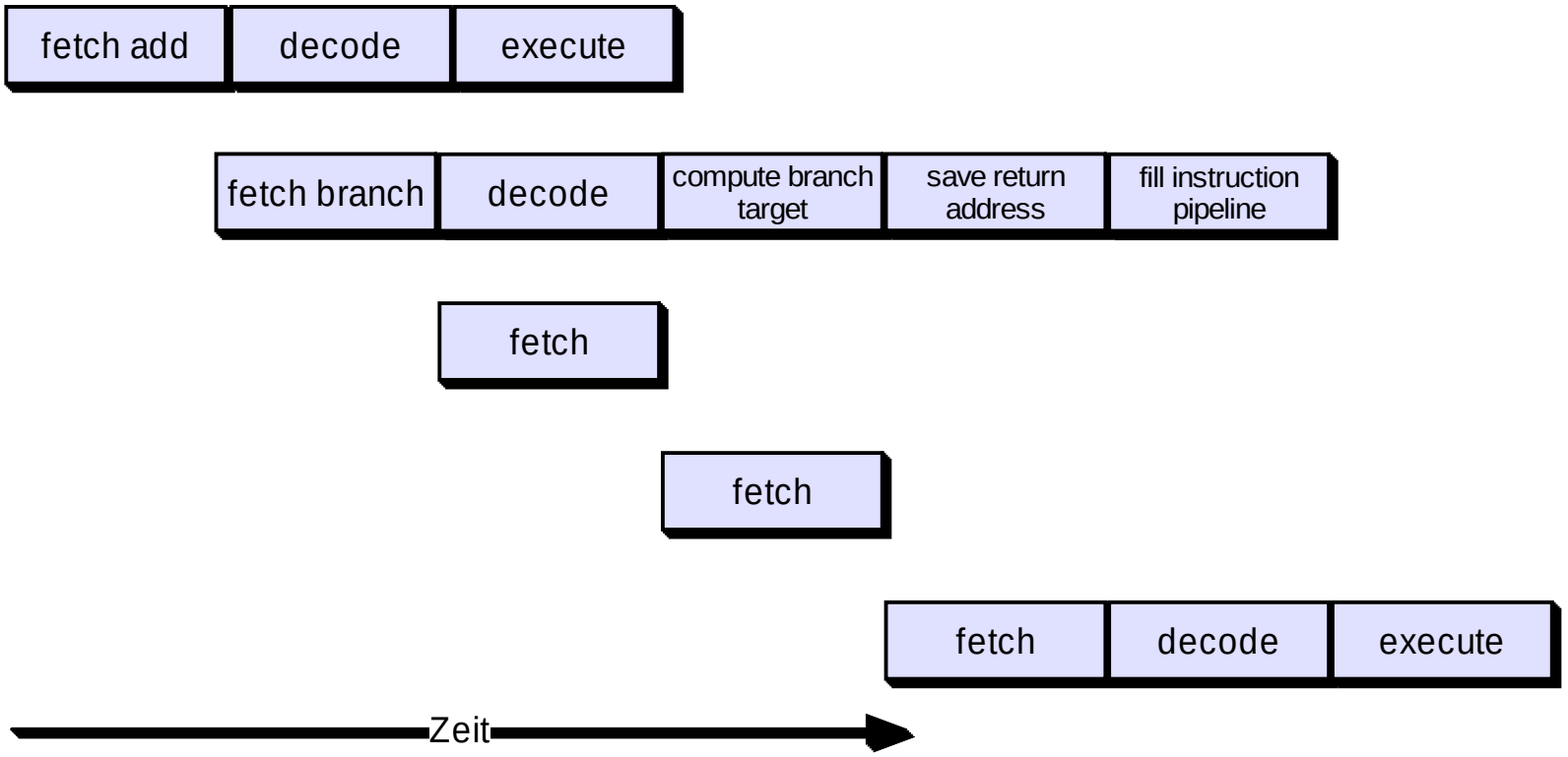




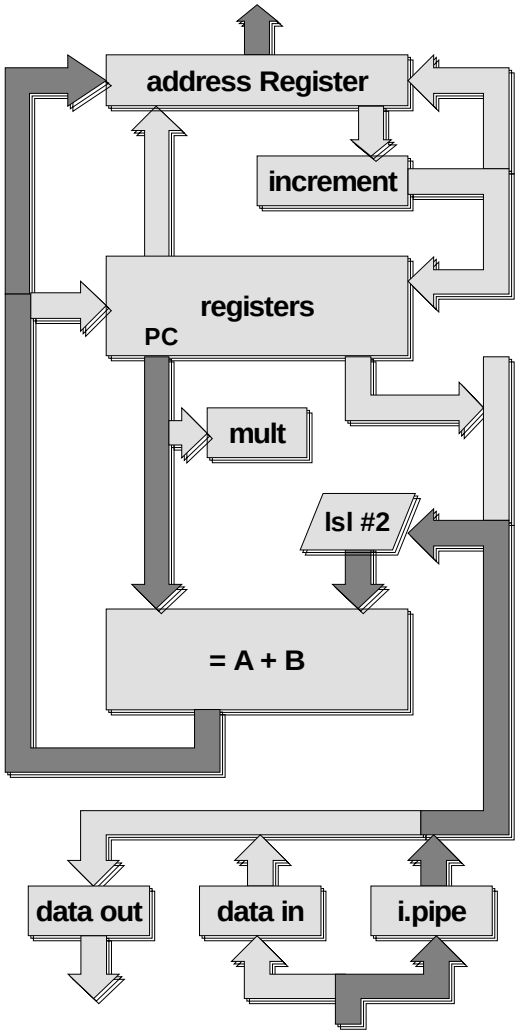
Store Register Datenpfad Aktivität 1. Zyklus berechne Adresse

Store Register Datenpfad Aktivität 2. Zyklus speichere Daten und auto index

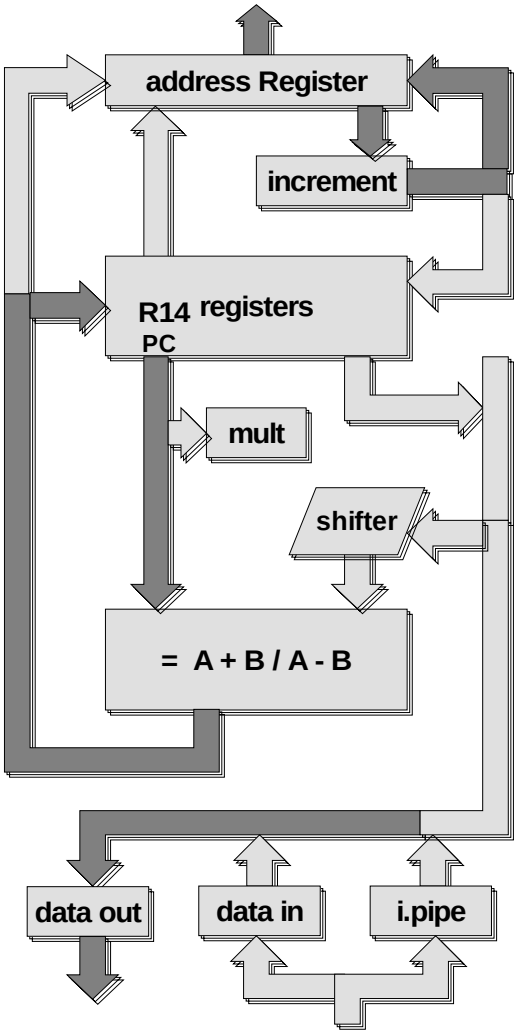
Pipeline für Branch Befehl



Branch Instruktion



Branch Instruktion 1. Zyklus
berechne Adresse



Branch Instruktion 2. Zyklus
speichere Return Adresse

RISC / CISC (1)

CISC: **C**omplex **I**nstruction **S**et **C**omputer

z.B. x86

CPU hat üblicherweise wenige Register

sehr viele Befehle, z.T. mächtige Befehle

ungleiche Befehlslängen, kompakter Code

Realisierung der Befehle durch Microcode

Zerlegung in kleinere Einheiten („Interpreter“ auf CPU)

Weiterentwicklung erzeugt komplexere Designs

RISC: **R**educed **I**nstruction **S**et **C**omputer

z.B. ARM, PowerPC

CPU hat viele Register

geringe Anzahl von Befehlen

einheitliches Datenformat (gleiche Befehlslänge), grösserer Code

direkte Verdrahtung der Befehle im Dekodierer

schnellere Ausführung

RISC / CISC (2)

Die Daten vom Speicher zu holen dauert.

Speicherbausteine sind langsam im Vergleich zu einem Prozessor.

Je mehr und je komplexere Befehle ein Prozessor hat desto länger dauert das Dekodieren

Wodurch wird RISC schnell ?

RISC: Speicherzugriffe minimieren

viele Register, Registerbefehle sind schnell

Load/Store Architektur (Zugriff auf Speicher **nur** über LDR/STR)

einheitliches Datenformat (4 Byte) erleichtert Dekodieren

günstig für Pipelining (nächste Befehl garantiert 4 Byte weiter)

Heutige Prozessoren meistens eine Kombination von RISC und CISC

Ausblick Mikroprozessorsysteme 2

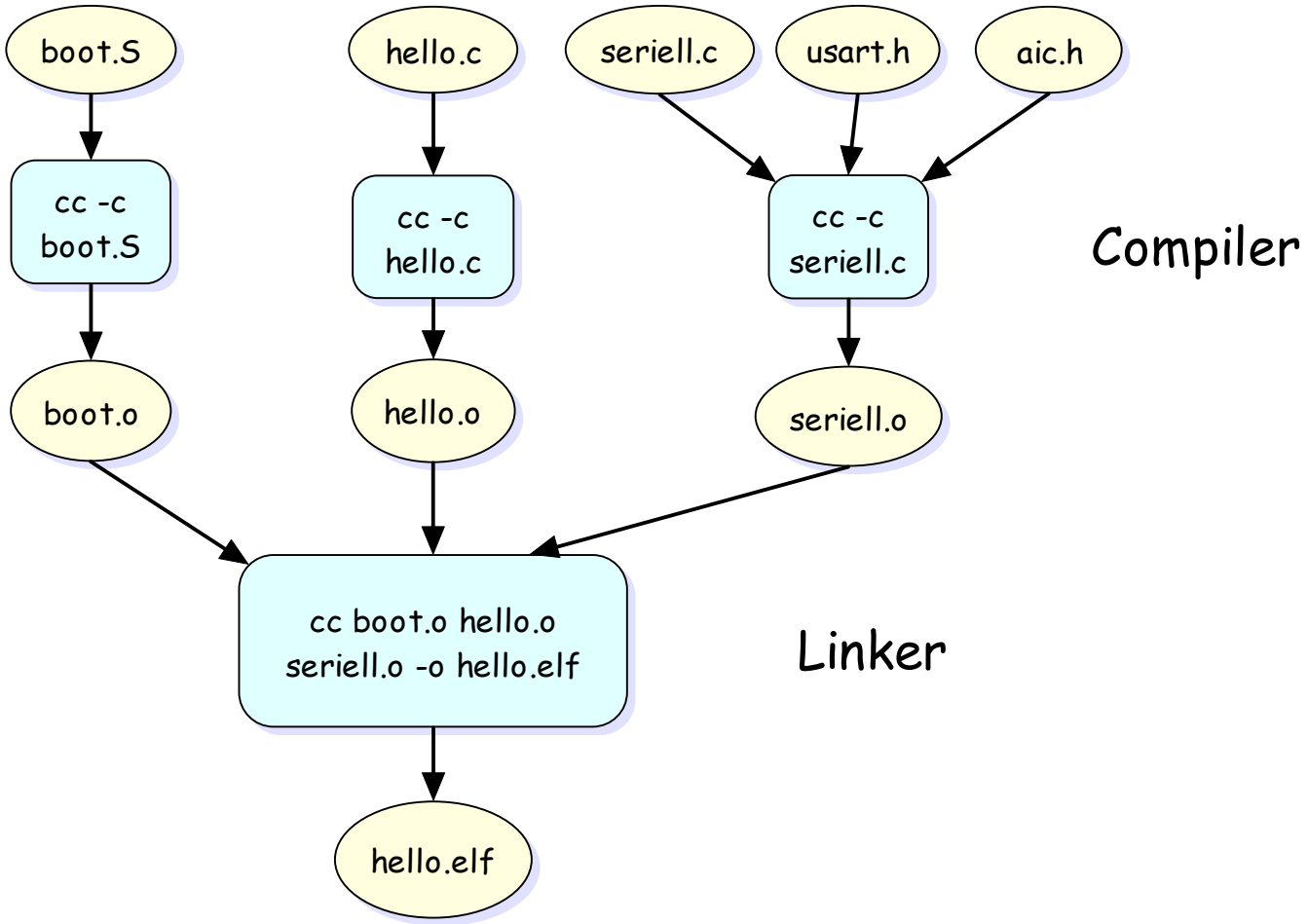
Schwerpunkt Vorlesung:

- Peripherie des ARM Prozessors
  - Power Management
  - Parallele I/O
  - Timer
  - Interrupt Handling
  - Serielle Schnittstelle
  - Softwareinterrupt

Praktikum:

- Implementierung einer Ausschankstation mit Waage und Pumpe
  - C-Programmierung mit geringem Assembleranteil

Compiler, Assembler und Linker



Compiler Flags

- CC -E hello.c
  - Aufruf des Preprozessors, Ausgabe auf STDOUT
- CC -S hello.c
  - Aufruf von Preprozessor und Compiler
  - Erzeugung von hello.s
- CC –c hello.c
  - Aufruf von Preprozessor, Compiler und Assembler
  - Erzeugung von hello.o
- CC hello.c
  - Aufruf von Preprozessor, Compiler; Assembler und Linker
  - Erzeugung von a.out
- CC hello.c -o hello.elf
  - Erzeugung von hello.elf

Compiler Flags

- CC -c boot.S
  - Aufruf von Preprozessor und Assembler
- CC -c hello.s
  - Aufruf nur des Assemblers, wenn der Dateityp *.s ist
- CC -c -O2 hello.c
  - Aufruf des Compilers mit Optimierungsstufe 2
- CC -c -g hello.c
  - Aufruf des Compilers und Erzeugung von Debuginformationen
- CC -c -I../myIncludes hello.c
  - Aufruf des Compilers, Suche nach Includefiles in ../myIncludes

Makefiles

- Makefiles dienen der automatischen Erstellung von Programmen aus einer Gruppe von Files
- Zum Erzeugen eines Programms, wird jeweils verglichen, ob ein Ziel älter ist als eine seiner Abhängigkeiten. In diesem Fall wird das Ziel neu gebildet
- Makefiles bestehen aus Regeln, wie ein Ziel aus seinen Abhängigkeiten erzeugt werden kann
- Syntax der Regeln:

```
ziel:        abhänigkeiten ...
        befehl1
        befehl2
        ...
```

Ein einfaches Makefile (1)

```
hello.elf: boot.o hello.o seriell.o
    arm-elf-gcc -e _start boot.o hello.o seriell.o -o hello.elf
hello.o: hello.c
    arm-elf-gcc -c -g  -O2 hello.c
boot.o: boot.S ../boot/ebi.inc
    arm-elf-gcc -c -g -I../boot boot.S
seriell.o: seriell.c ../h/usart.h ../h/aic.h
    arm-elf-gcc -c -g -I../h seriell.c
```

Der Name von Makefiles

- Makefiles sollten entweder makefile oder Makefile genannt werden
- Der Name des Makefiles kann auch explizit über den -f Switch vorgegeben werden
- Findet make kein Makefile, so muß ihm ein Ziel vorgegeben werden. Make versucht dann mit Hilfe von impliziten Regeln das Ziel zu bilden

Variablen in Makefiles

- Makefiles können durch die Definition von Variablen einfacher und flexibler gestaltet werden

```
CC := arm-elf-gcc
INCLUDE := ../h
BOOT := ../boot
CFLAGS := -c -g -O2
LDFLAGS := -e _start
```

Makefile mit Variablen (2)

```
CC := arm-elf-gcc
INCLUDE := ../h
BOOT := ../boot
CFLAGS := -c -g -O2
LDFLAGS := -e _start

hello.elf: boot.o hello.o seriell.o
    $(CC) $(LDFLAGS) boot.o hello.o seriell.o -o hello.elf
hello.o: hello.c
     $(CC) $(CFLAGS) hello.c
boot.o: boot.S $(BOOT)/ebi.inc
     $(CC) $(CFLAGS) -I $(BOOT) boot.S
seriell.o: seriell.c $(INCLUDE)/usart.h $(INCLUDE)/aic.h
     $(CC) $(CFLAGS) -I$(INCLUDE) seriell.c
```

automatische Variablen und VPATH

- Make kennt einige automatische Variablen, die die Schreibweise sehr vereinfachen

  $@ das Ziel einer Regel

  $^ die Liste der Abhängigkeiten, durch Blanks getrennt

  $< die erste Abhängigkeit in der Liste der Abhängigkeiten

- Die Variable VPATH definiert einen Suchpfad für Dateien in der Abhängigkeitsliste

  VPATH := $(INCLUDE):$(BOOT)

Makefile 3

```
CC := arm-elf-gcc
INCLUDE := ../h
BOOT := ../boot
VPATH := $(INCLUDE):$(BOOT)
CFLAGS := -c -g -O2
LDFLAGS := -e _start

hello.elf: boot.o hello.o seriell.o
    $(CC) $(LDFLAGS) $^ -o $@
hello.o: hello.c
     $(CC) $(CFLAGS) $<
boot.o: boot.S ebi.inc
     $(CC) $(CFLAGS) -I $(BOOT) $<
seriell.o: seriell.c usart.h aic.h
     $(CC) $(CFLAGS) -I$(INCLUDE) $<
```

Suffix Regeln

```
CC := arm-elf-gcc
INCLUDE := ../h
BOOT := ../boot
VPATH := $(INCLUDE):$(BOOT)
CFLAGS := -c -g -O2
LDFLAGS := -e _start

.c.o:
	$(CC) $(CFLAGS) -I$(INCLUDE) $<
hello.elf: boot.o hello.o seriell.o
	$(CC) $(LDFLAGS) $^ -o $@
hello.o: hello.c
seriell.o: seriell.c usart.h aic.h
boot.o: boot.S ebi.inc
	$(CC) $(CFLAGS) -I $(BOOT) $<
```

Wildcard Beispiel

- Eine Regel zum Löschen aller Objekt Files

```
clean:
        rm -f *.o
```

- Eine Regel zum Drucken aller geänderten .c Files

```
print: *.c
        lpr -p $?
        touch print
```

- Aufruf make print führt den Befehl `lpr' aus, falls sich eine Quelldatei seit dem letzten Aufruf geändert hat
- Die automatische Variable `$?' wird benutzt um (nur) die geänderten Dateien zu drucken