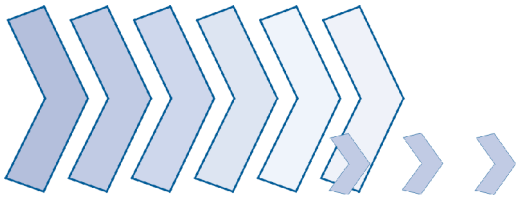
HUMBOLDT-UNIVERSITÄT ZU BERLIN



Technische Informatik 2
Maschinenprogram-mierungskonzepte

Prof. Dr. Miroslaw Malek

Sommersemester 2007

www.informatik.hu-berlin.de/rok/ca

Thema heute

- Ausführung von Befehlen
- Ein-/Ausgabeprogrammierung
- Architekturen
  - Stackspeichermaschinen
  - Allzweck-Register
- Unterprogrammaufrufe

Wahl der Sprache

- ML – Machine Language

  Maschinensprache (Benutzung von 0 und 1 oder Hexadezimalzahlen, unhandlich und fehleranfällig)
- AL – Assembly Language

  Assembler (maschinenabhängig, aber effizient)
- HLL – High-Level Language

  Hochsprache (handlich und universell)

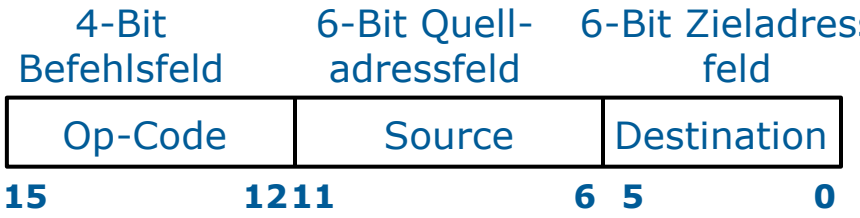
Compiler/Assembler und Betriebssystem in Kombination mit der Rechnerarchitektur -Befehlssatz und E/A Fähigkeiten- sind entscheidende Faktoren bezüglich der Architektursperformance

Warum eigentlich Assembler?

Die wichtigsten Vorteile der Kenntnis von Assembler:

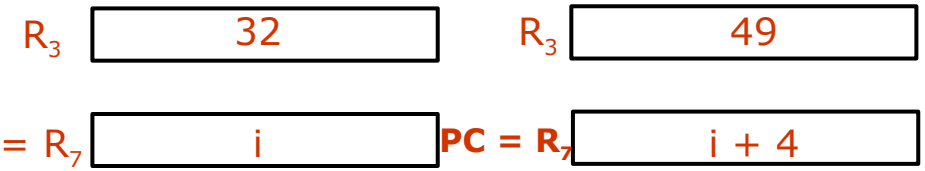
- Verständnis, wie ein Rechner arbeitet
- Ermöglicht die Programmierung von eingebetteten Systeme mit beschränkten Ressourcen
- Debugging: Wie werden Programme “entwanzt”? Wie werden Fehler gesucht?
- Ermöglicht hardwarenahe Performance-optimierung

Beispiel einer Befehlsausführung

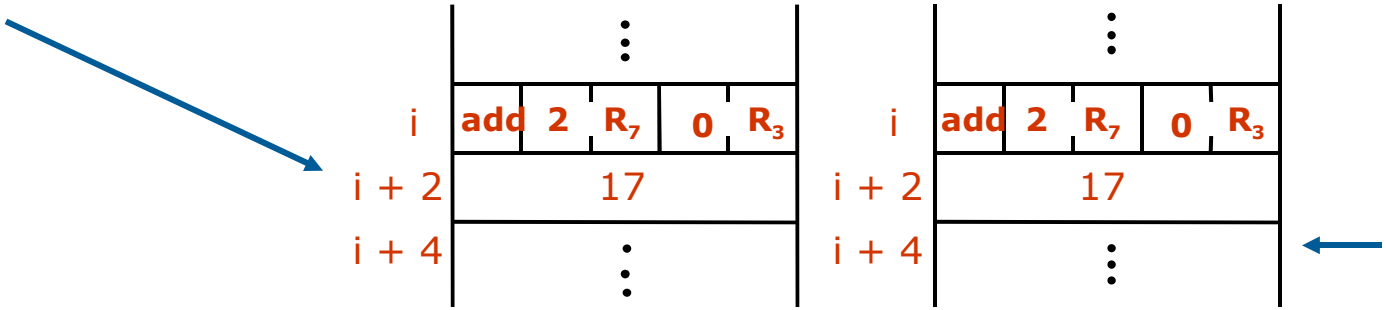


ADD #R7, R3

Allgemeines Format



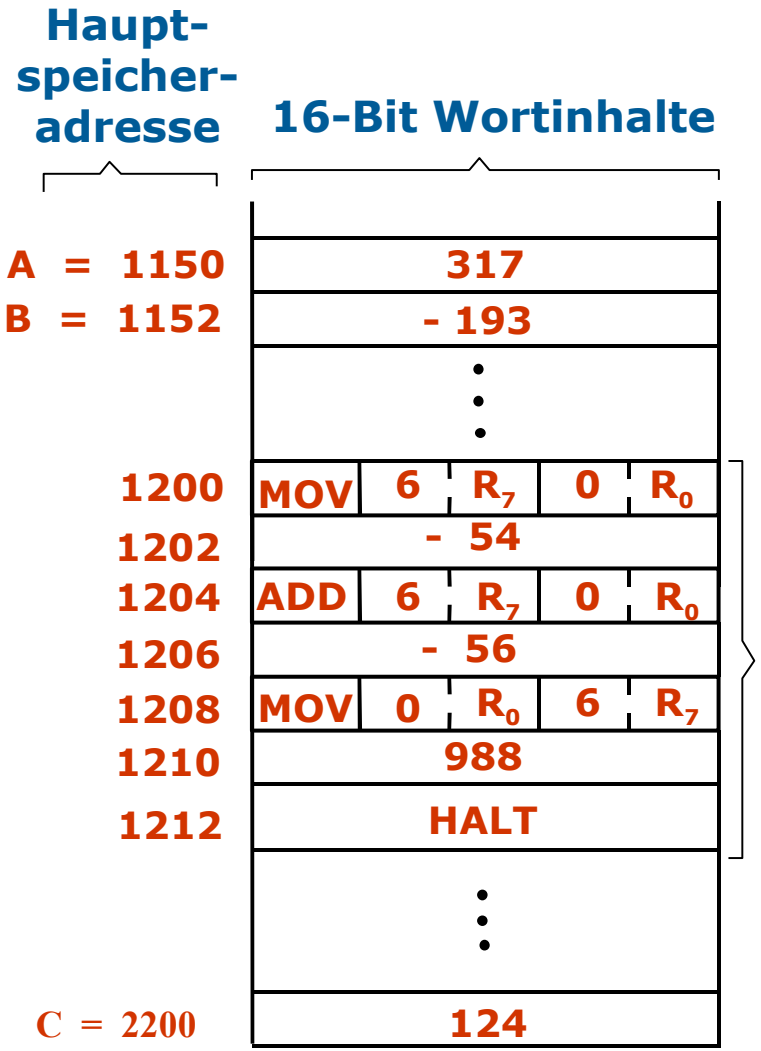
Autoincrement mode 010 (2_{10}): EA=[PC]=[R_7] und increment PC=R_7



Bevor Befehls-ausführung

Nach Befehls-ausführung

Beispiel einer Befehlsausführung (2)



1) $R_0 = 317$

2) $R_0 = 124$

EA = X + PC = -54 + 1204 = 1150

```
MOV   A, R0
ADD   B, R0
MOV   R0, C
HALT
```

Assemblerversion des Programms

Modus "6" - Relative Indexadressierung

EA=X + [PC]=X + [R_7]

Beispiel der Ausführung des PowerPC Befehls StoreWord

Autoincrement Modus stw: stw rS, d(rA),

wobei rS (Quelle)und rA (Ziel) Register sind und d ein Offset ist.

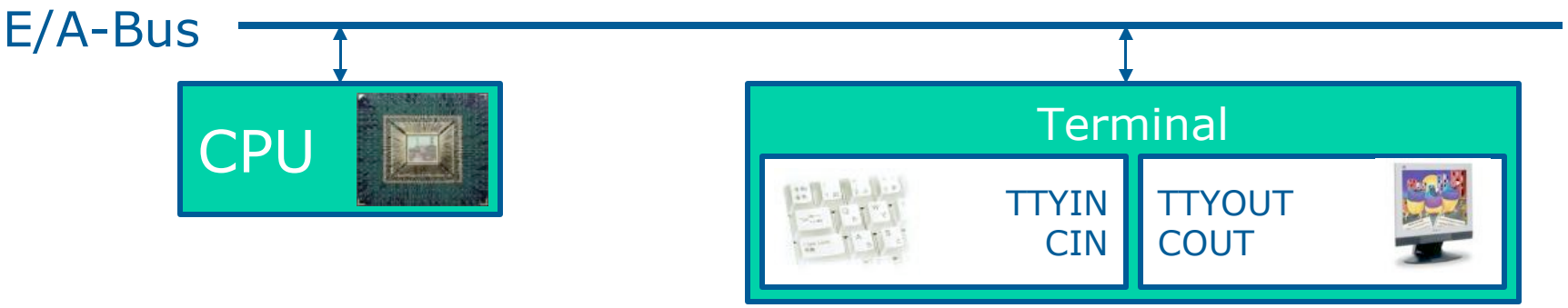
Der Inhalt von rS wird in die Speicherzelle geschrieben, die durch EA bestimmt ist.

Dabei gilt: EA=(rA)+d.

Binäre Darstellung des Mnemonics stw: 100100

Dezimale Darstellung des Mnemonics stw: 36

Ein-/Ausgabeprogrammierung



- TTYIN — 8-Bit-Pufferregister der Tastatur zugeordnet
- TTYOUT — 8-Bit-Pufferregister dem Bildschirm zugeordnet
- CIN, COUT — Steuer-Flags
  - CIN = 1 — informiert CPU, dass ein gültiges Zeichen in TTYIN ist
  - COUT= 1 — informiert CPU, ein gültiges Zeichen nach TTYOUT zu schicken
- MOVB TTYIN, R1 — Eingabeoperation
- MOVB R1, TTYOUT — Aufzeichnung

Ein-/Ausgabeprogrammierung (2)

	MOV	#LOC,R0	Initialisiere das Zeigerregister R0 mit der ersten Position des Hauptspeicherbereichs, in den die Zeichen geladen werden sollen.
READ:	TSTB	KBSTATUS	Warte auf das Zeichen, das in das Tastaturpufferregister TTYIN eingegeben wird
	BPL	READ	
	MOVB	TTYIN,@R0	Übertrage das Zeichen aus dem TTYIN in den Hauptspeicher (CIN wird auf 0 gesetzt)

Ein-/Ausgabeprogrammierung (3)

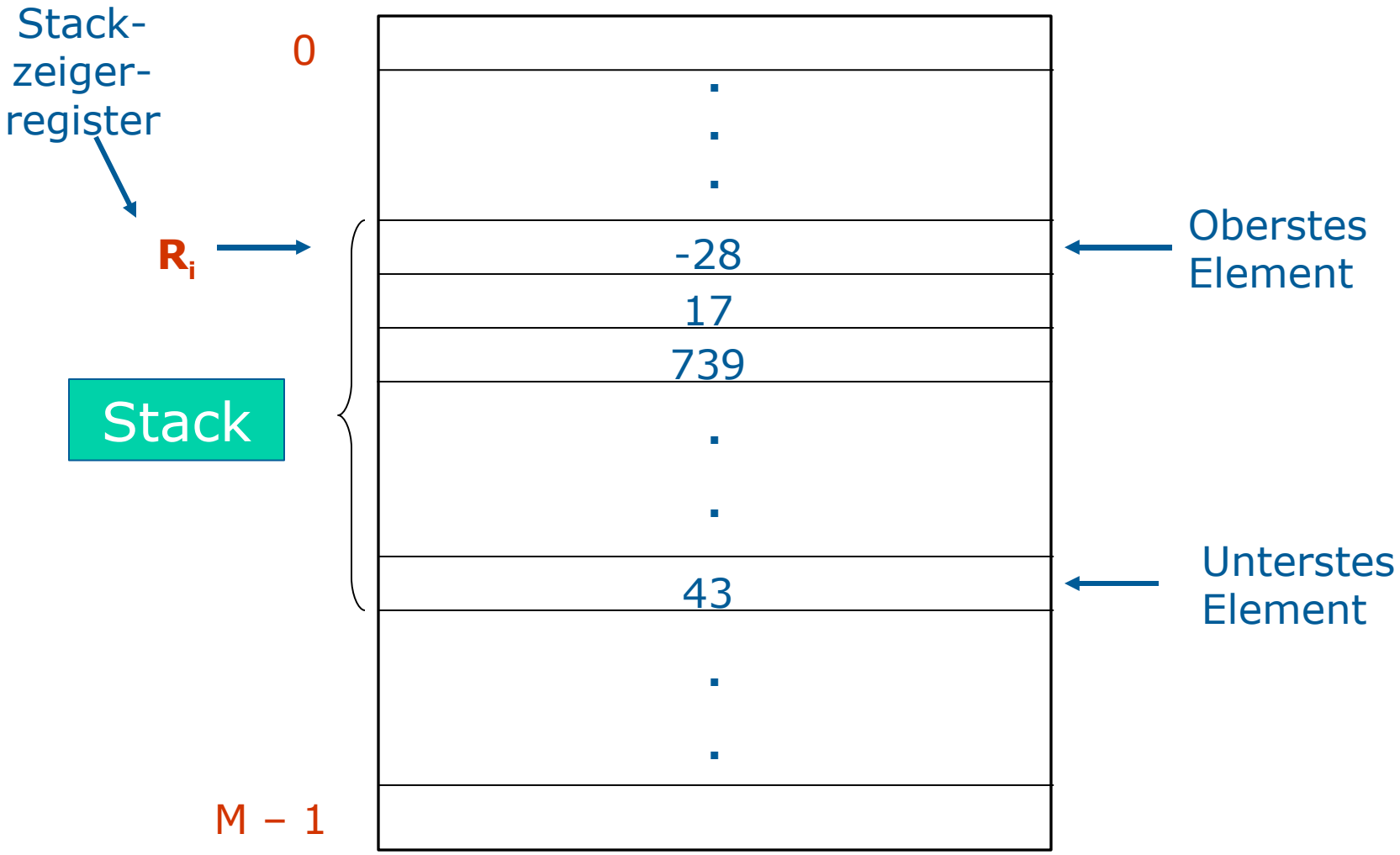
```
ECHO-
BACK: TSTB   PRSTATUS      Warte bis der Bildschirm bereit ist.
      BPL    ECHOBACK
      MOVB   @R0,TTYOUT    Verschiebe das gerade gelesene
                           Zeichen als Ausgabe in das
                           Bildschirmpufferregister TTYOUT
                           (COUT wird auf 0 gesetzt)

      CMPB   (R0)+,#CR     Überprüfe, ob das gerade
      BNE    READ          eingelesene Zeichen CR
      HALT                 (Wagenrücklauf) ist. Falls nicht,
                           springe zurück und lese das
                                 nächste Zeichen;
                                       an. Das Zeigeregister
                                             inkrementiert,
   anderenfalls halte
   wird
   wenn ein weiteres
   Zeichen gelesen wurde.
```

Architekturtypen

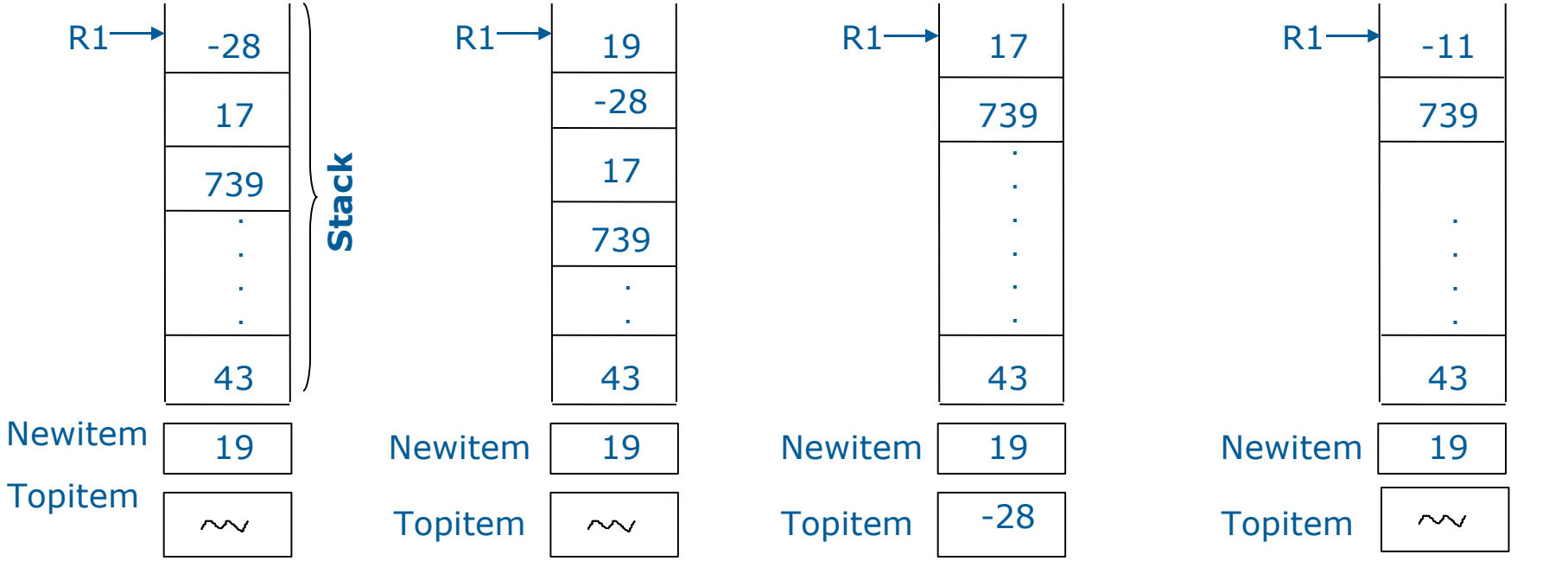
- Stack- Architekturen
- Register-Register Architekturen
- Register-Speicher Architekturen
- Speicher - Speicher Architekturen
- Markierte Architekturen (tagged architectures)
- Tradeoffs
  - Leistung
  - Effizienz
  - Design Komplexität
  - Einfachheit von Programmierung
  - Einfachheit der Parameterübergabe und der Unterprogrammaufrufe
  - Rekursion

Ein Stack im Hauptspeicher



Beispiele für Stackoperationen

MOV NEWITEM,- (R1) **(push)**
MOV (R1)+,TOPITEM **(pop)**
(Bemerkung (a) => (b), (a) => (c), (a) => (d))



(a)Initiale Inhalte des Stacks

(b) Nach Push von NEWITEM

c) Nach Pop zum TOPITEM

d) Nach Ausführung ADD (R1)+, @R1

(Stack-)Stapelspeichermaschinen

- Die meisten Befehle beziehen sich auf die oberen Einträge (meistens die oberen 2) eines Stapelspeichers
- Um Daten zwischen dem Speicher und dem Top of Stack zu verschieben werden zusätzliche Befehle bereitgestellt
- Die oberen Einträge des Stapelspeichers (2 bis 8 oder mehr) werden in der CPU festgehalten
- Befehle beziehen sich implizit auf den Top of Stack
- Ideal um Ausdrücke zu berechnen (Stapelspeicher hält dazwischenliegende Resultate)
- Werden als gute Architektur in Zusammenhang mit höheren Programmiersprachen angesehen

Stapelspeichermaschinen (2)

- Nachteile:
  - werden sehr langsam, wenn der Stapelspeicher größer als der lokale Speicher der CPU wird
  - kein einfacher Zugriff auf Daten in der Mitte des Stapelspeichers

- Binärarithmetische und logische Operationen:
  - Operanden: obere 2 auf dem Stapelspeicher
  - Operanden werden aus dem Stapelspeicher entfernt
  - das Ergebnis wird auf dem Top of Stack platziert

Stapelspeichermaschinen (3)

- Unärarithmetische und logische Operationen:
  - Operand auf dem Top of Stack
  - Operand wird durch das Ergebnis der Operation ersetzt

- Datenverschiebeoperationen:
  - push: platzieren eines Speicherinhalts auf dem Top of Stack
  - pop: verschieben von Top of Stack in den Speicher

Stapelspeichermaschinen: Programmbeispiele

Wir wählen unseren bevorzugten Ausdruck ($y \leftarrow ax^2 + bx + c$) aus; wir benutzen eine hypothetische Assemblersprache (tos = Top of Stack):

push	a	tos: a
push	x	tos: a x
dup		tos: a x x
mult		tos: a x^2
mult		tos: a x^2
push	b	tos: a x^2b
push	x	tos: a x^2 b x
mult		tos: a x^2 bx
push	c	tos: a x^2 bx c
add		tos: a x^2 bx+c
add		tos: a x^2+bx+c
pop	y	y ← a x^2+bx+c

General Purpose Register Maschinen (GPRM)

- Bei Stapelspeichermaschinen sind nur die obersten zwei Elemente des Stapels direkt für Befehle verfügbar. In General Purpose Register Maschinen ist der CPU-Speicher als ein Satz von Registern organisiert, die für die Befehle gleichermaßen verfügbar sind
- Wiederholt benutzte Operanden werden in Register plaziert (unter Aufsicht des Programms)
- dies reduziert die Befehlsgröße
- ebenso werden Speicherzugriffe reduziert

GPR-Maschinen: Programmbeispiel

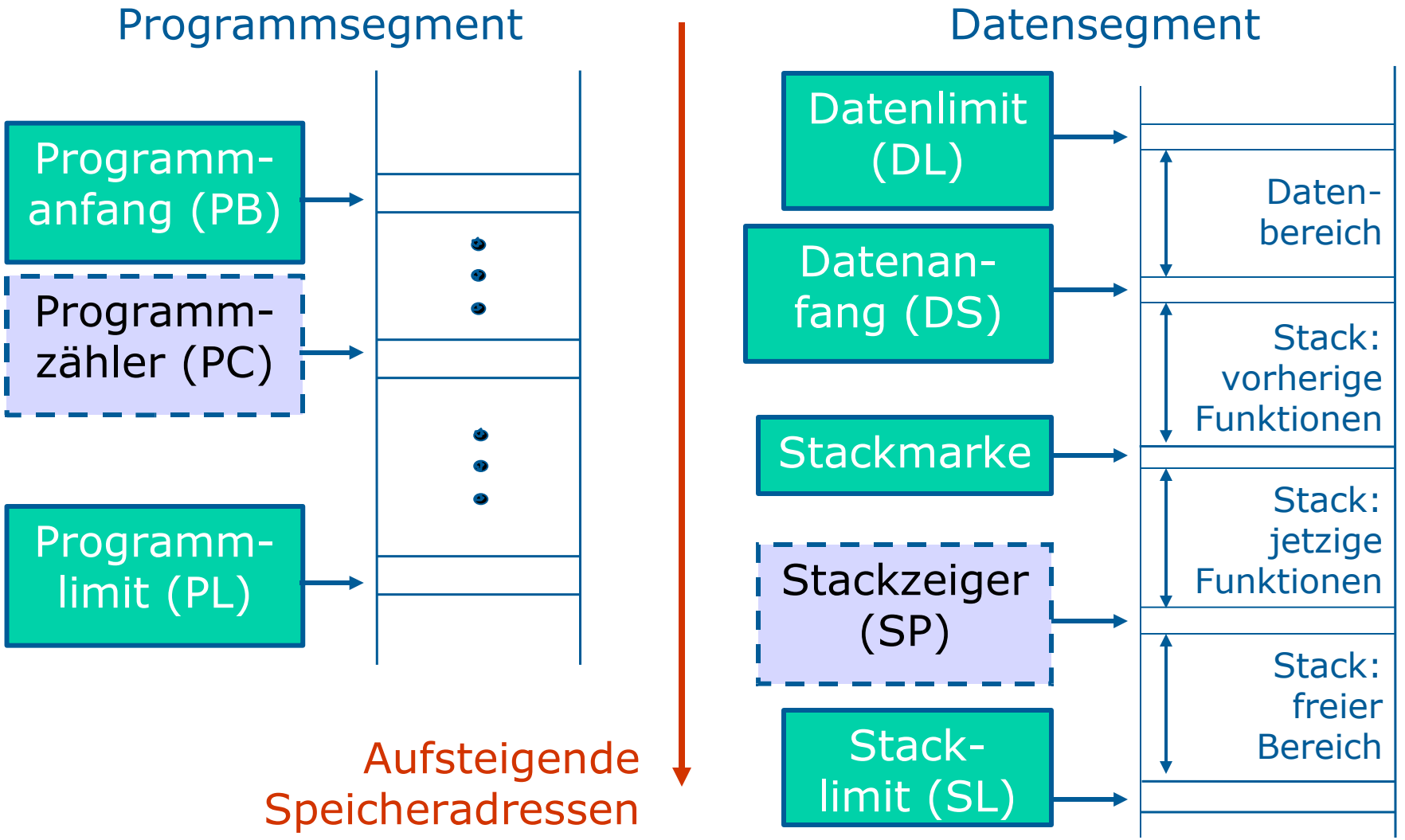
Wieder werten wir ($y \leftarrow ax^2 + bx + c$) auf einer hypothetischen Maschine mit 16 Registern R0 bis R15 und 2 Operanden-Registern aus

load	x, R1	R1 ← x
load	a, R2	R2 ← a
load	b, R3	R3 ← b
load	c, R4	R4 ← c
mult	R1, R2	R2 ← ax
mult	R1, R2	R2 ← ax^2
mult	R1, R3	R3 ← bx
add	R2, R3	R3 ← ax^2 + bx
add	R3, R4	R4 ← ax^2 + bx + c
store	R4, y	y ← ax^2 + bx + c

GPR-Maschinen: Beispiele

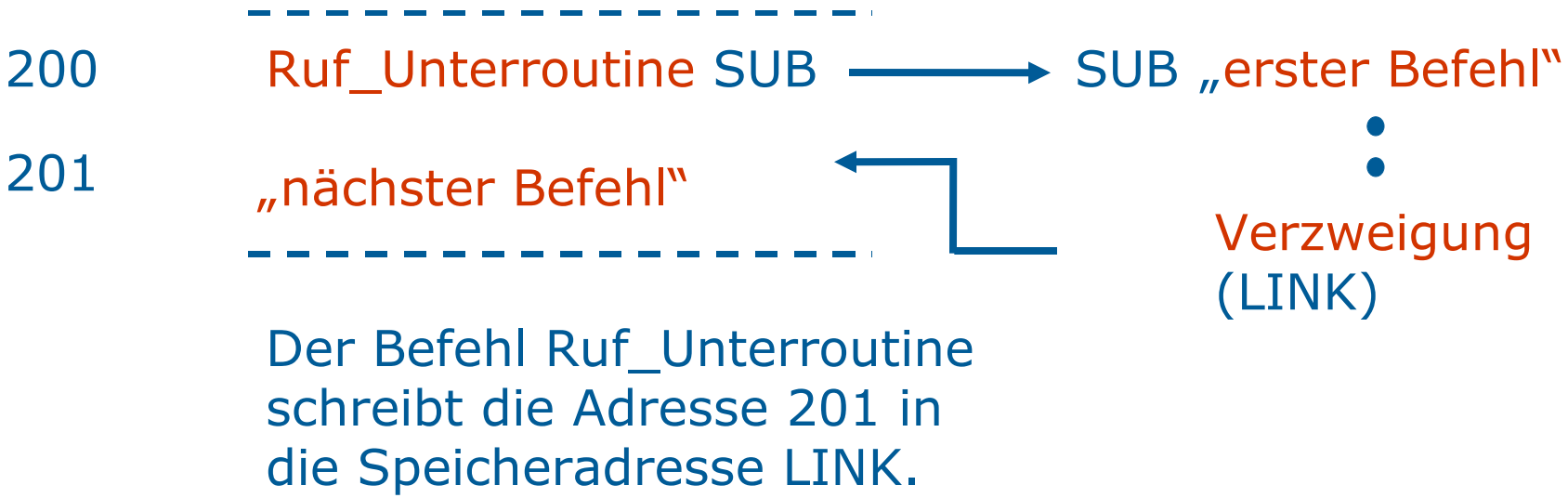
- PDP-11, LSI-11, VAX
- IBM 360, IBM 370, PC-RT, RISC 6000
- Motorola, all series
- CDC 6600, 7600, CYBER
- SPARC, MIPS, x86, PowerPC, Pentium, Itanium

Programm- und Datensegment-organisation: Beispiel MC68000



Programmaufruf mit Hilfe von Stack: 1. dedizierte Adresse

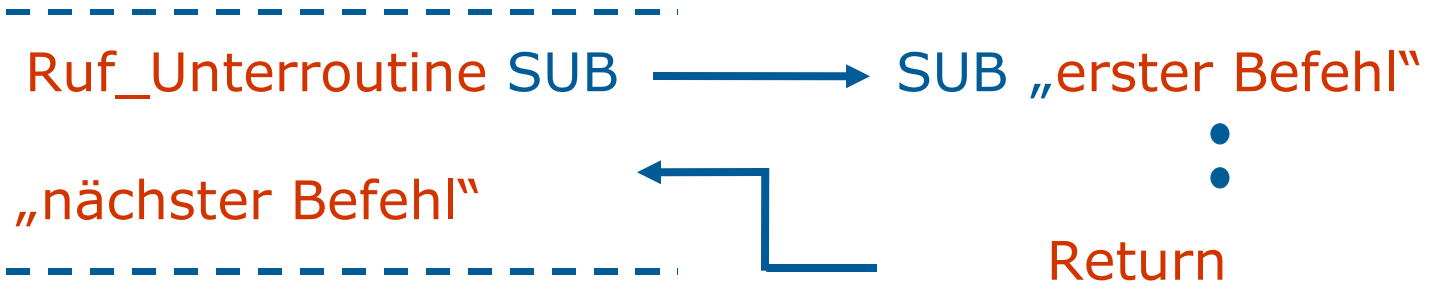
Speicher-adresse	Rufendes Programm	Unterroutine



Der Befehl Ruf_Unterroutine schreibt die Adresse 201 in die Speicheradresse LINK.

Programmaufruf mit Hilfe von Stack: 2. Benutzung von Stack

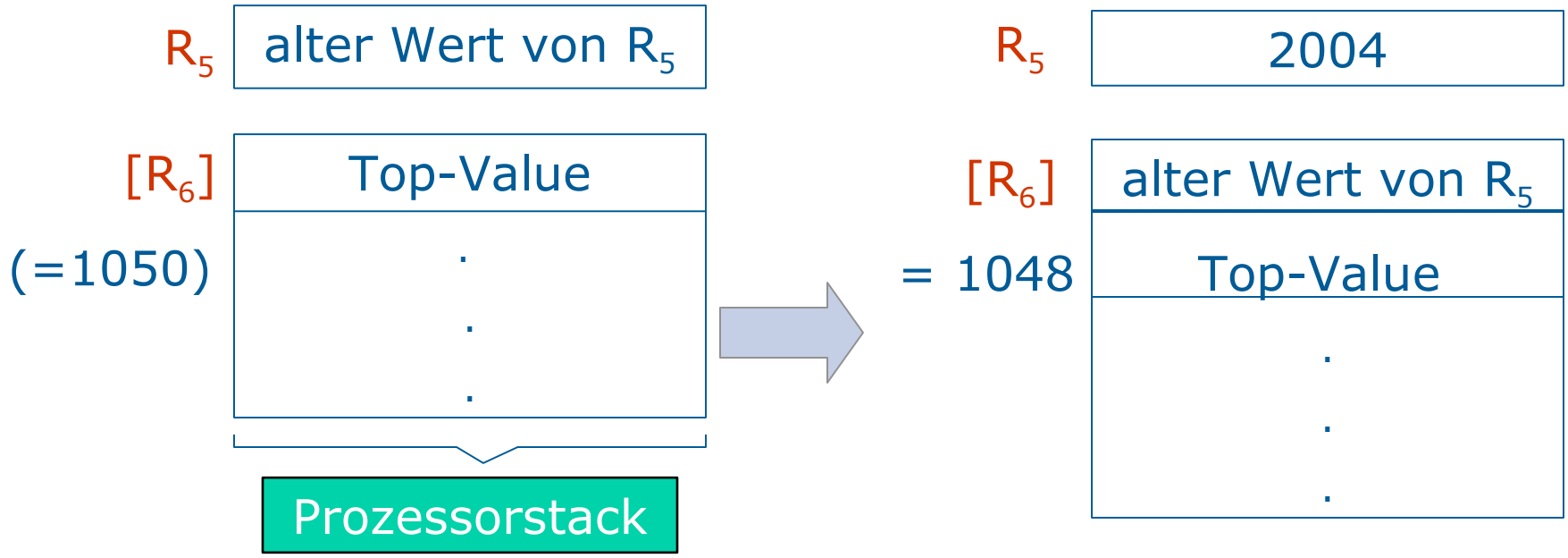
Speicher-adresse	Rufendes Programm	Unterroutine



Der Befehl Ruf_Unterroutine legt die Adresse 201 auf den Stack.

Der Rückkehrbefehl lädt den obersten Stackeintrag in den PC.

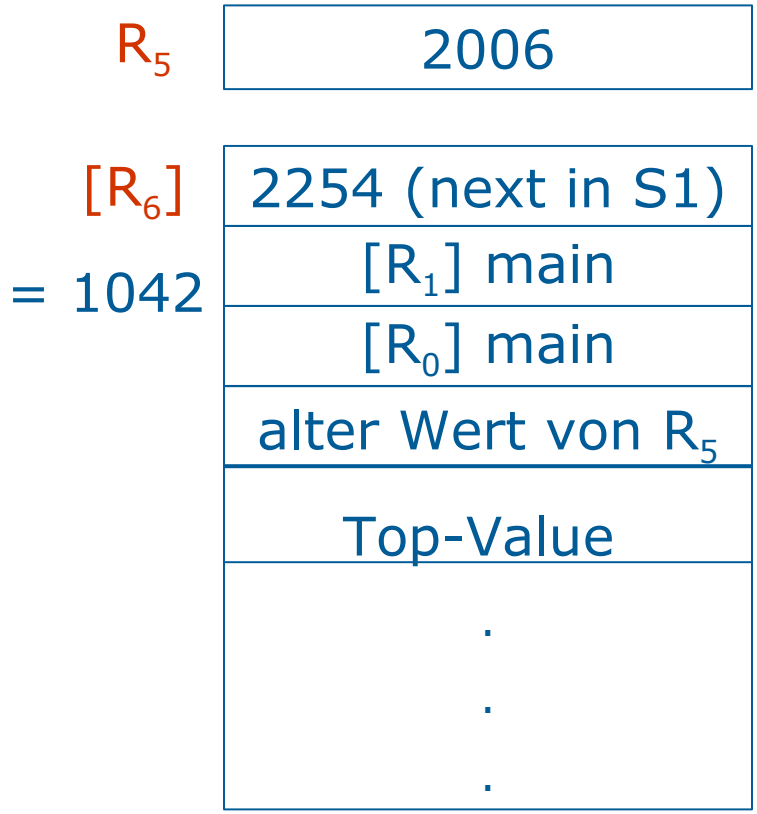
Aufruf eingebetteter Unterprogramme auf dem Stack



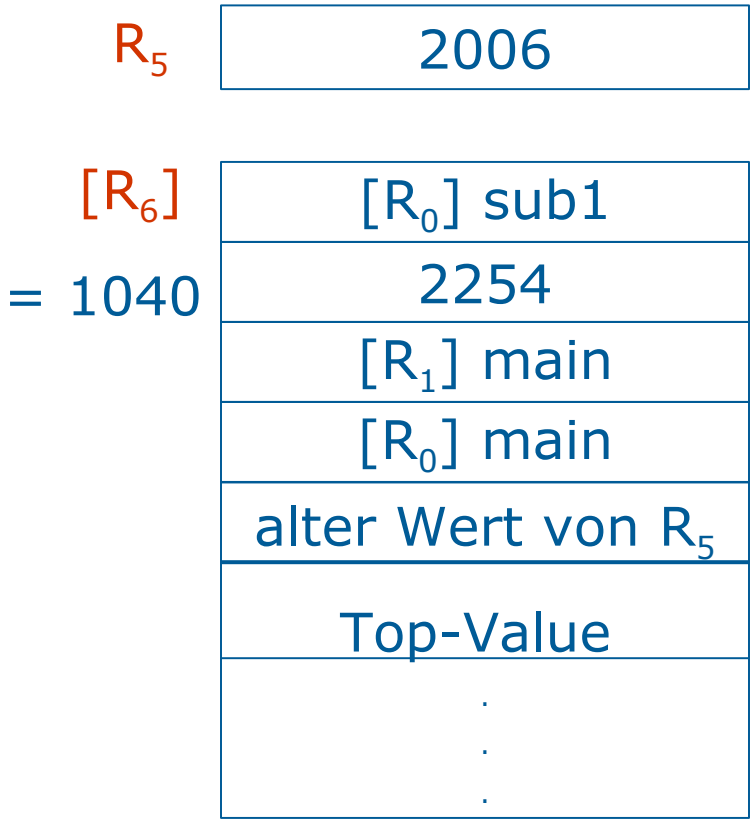
a. Initial: R_5 ist ein linkage register (speichert den Wert des PC). R_6 ist ein Zeiger zum Prozessorstack

a. Nach Ausführung von JSR R5, SUB1

Aufruf eingebetteter Unter-programme auf dem Stack (2)

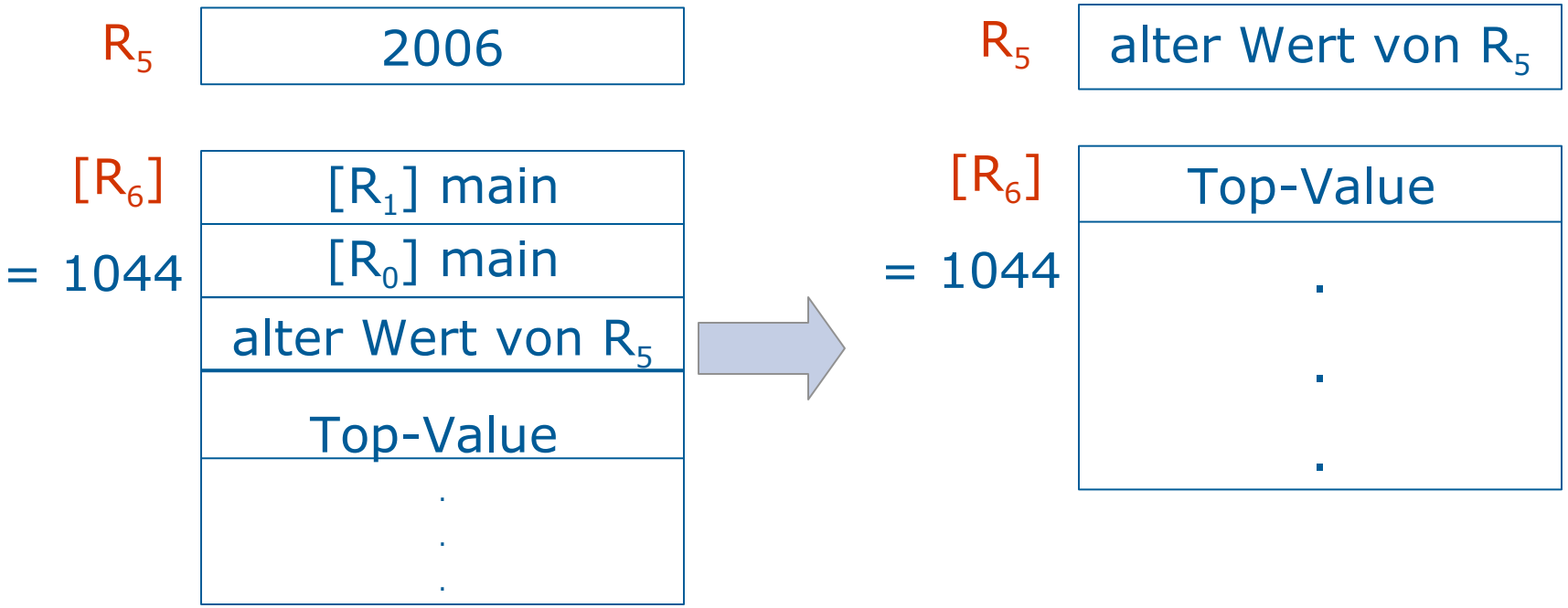


a. Nach Ausführung von
JSR R7, SUB2



a. Nach Ausführung von
MOV R0, -(R6) in SUB 2

Aufruf eingebetteter Unter-programme auf dem Stack (3)



a. Nach Ausführung von RTS R7

a. Nach Ausführung von RTS R5

Beispielprogramm für die Berechnung der Fakultät im PowerPC

```
        lbz r1, 15 (r0)
        li  r2, #1
fac:    cmpwi LT, r1, #2
        blt end
        mullw r2, r2, r1
        addi r1, r1, #-1
        b fac
end:    blr
```

Hole n von MEM[15+r0] & speichere in r1

Speichert 1 in das Register r2

Vergleiche r1 mit der Zahl 2; LT ist das Bit für „kleiner als“ im Conditionregister(CR). LT wird gesetzt wenn r1 < 2.

Bedingter Sprung zum Label FINISHED, wenn das LT Bit in CR gesetzt ist.

Multipliziere r2, mit r1, das Ergebnis wird in r2 gespeichert.

Dekrementiere r1 um 1

Springe zum Label FAC

Verzweige zum Link- Register, dort hat der Aufrufer eine Rücksprungadresse hinterlassen

Fragen?