Rechnerstrukturen, Teil 2

Vorlesung 4 SWS WS 19/20

Prof. Dr. Jian-Jia Chen
Fakultät für Informatik – Technische Universität Dortmund
jian-jia.chen@cs.uni-dortmund.de
http://ls12-www.cs.tu-dortmund.de

Kontext

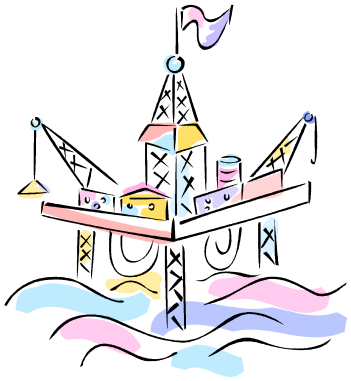
Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information
[Gesellschaft für Informatik]

Z.B. Zahlendarstellung in RS, Teil 1

hier und jetzt

Motivation

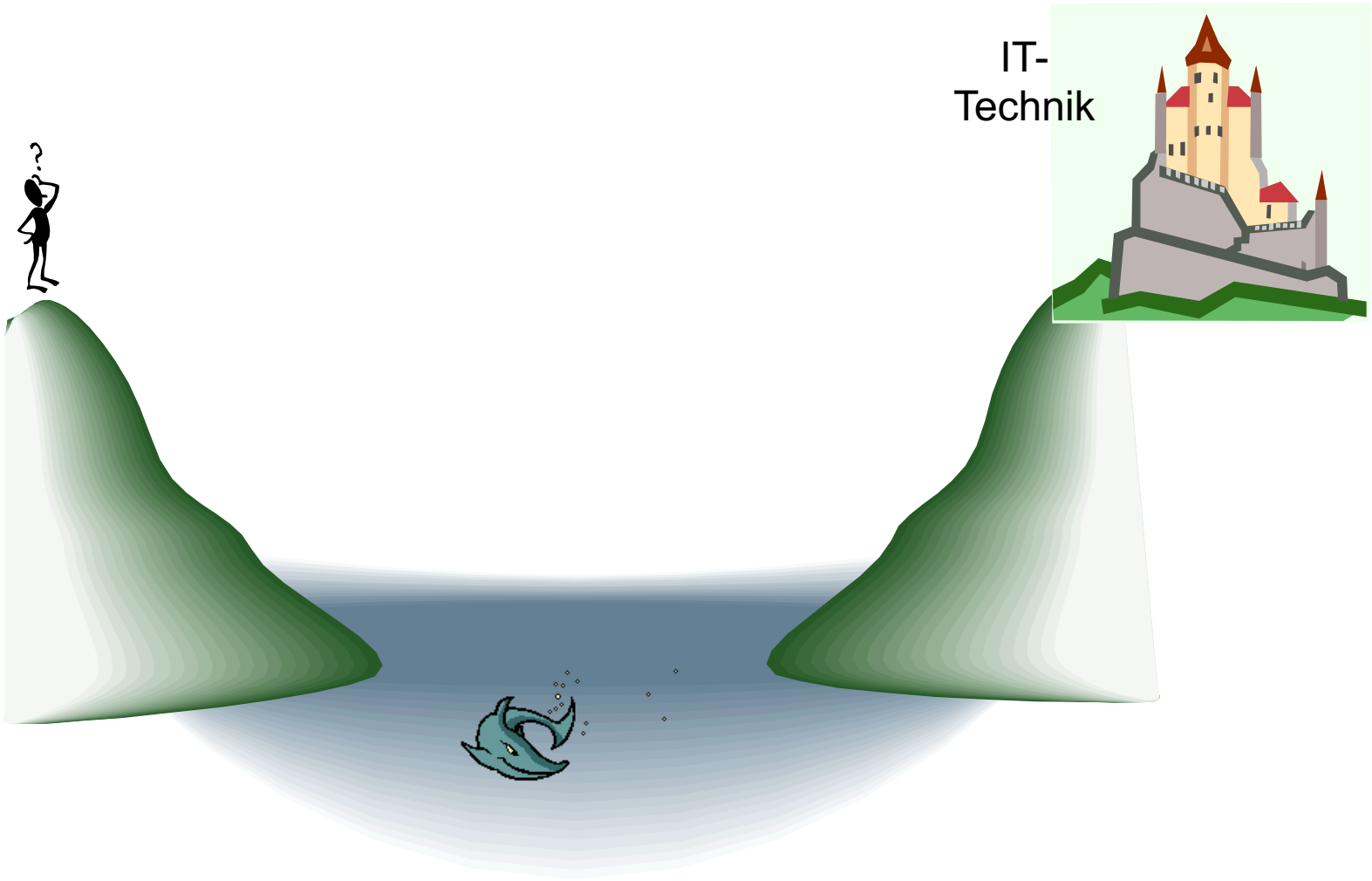
- Jede Ausführung von Programmen bedarf einer zur Ausführung fähigen Hardware.
- Wir nennen diese auch Ausführungsplattformen (***execution platforms)***.
- *Platform-based design* ist ein Ansatz für viele Anwendungen (Handys, Autos, …)
- Plattformen sind nicht immer ideal (z.B. führen Anwendungen nicht in 0 Zeit mit 0 Energie aus)
- Grundlegendes Verständnis für nicht-ideales Verhalten ist wichtig
- Deshalb Beschäftigung in dieser Vorlesung mit Ausführungsplattformen.



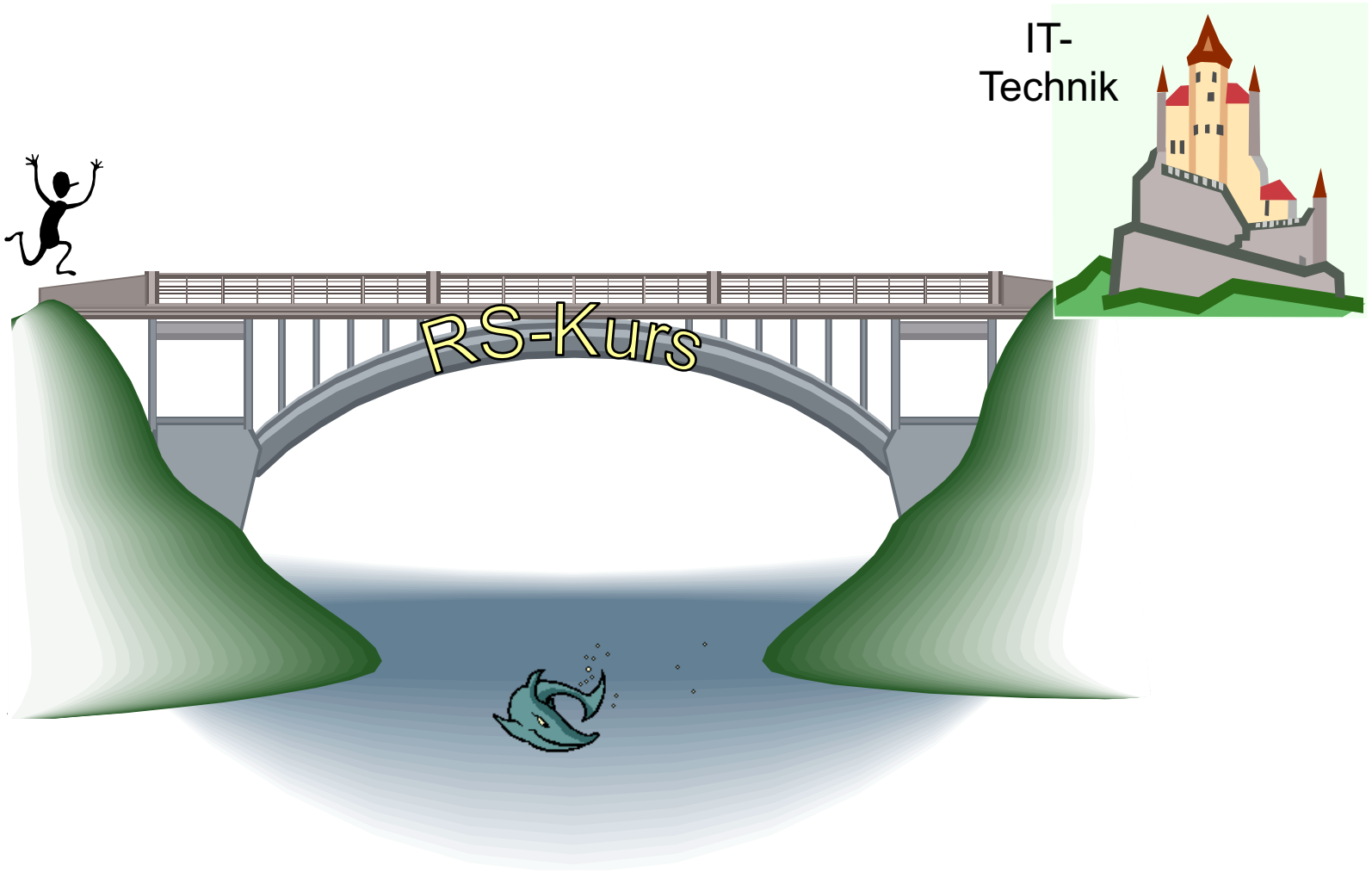
*

* Bei Graphiken ohne Quellenangabe handelt es sich um ClipArts der Fa. Microsoft, deren Nutzungseinschränkungen einzuhalten sind.

Problematische Situationen …



… und Techniken zu deren Vermeidung



Thema des zweiten Teils der Vorlesung

Ziel: Verständnis der Arbeitsweise von Rechnern, einschl.

- der Programmierung von Rechnern
- des Prozessors
- der Speicherorganisation
- des Anschlusses von Peripherie
- Anwendungen bei Eingebetteten Systemen

Stil des zweiten Teils der Vorlesung

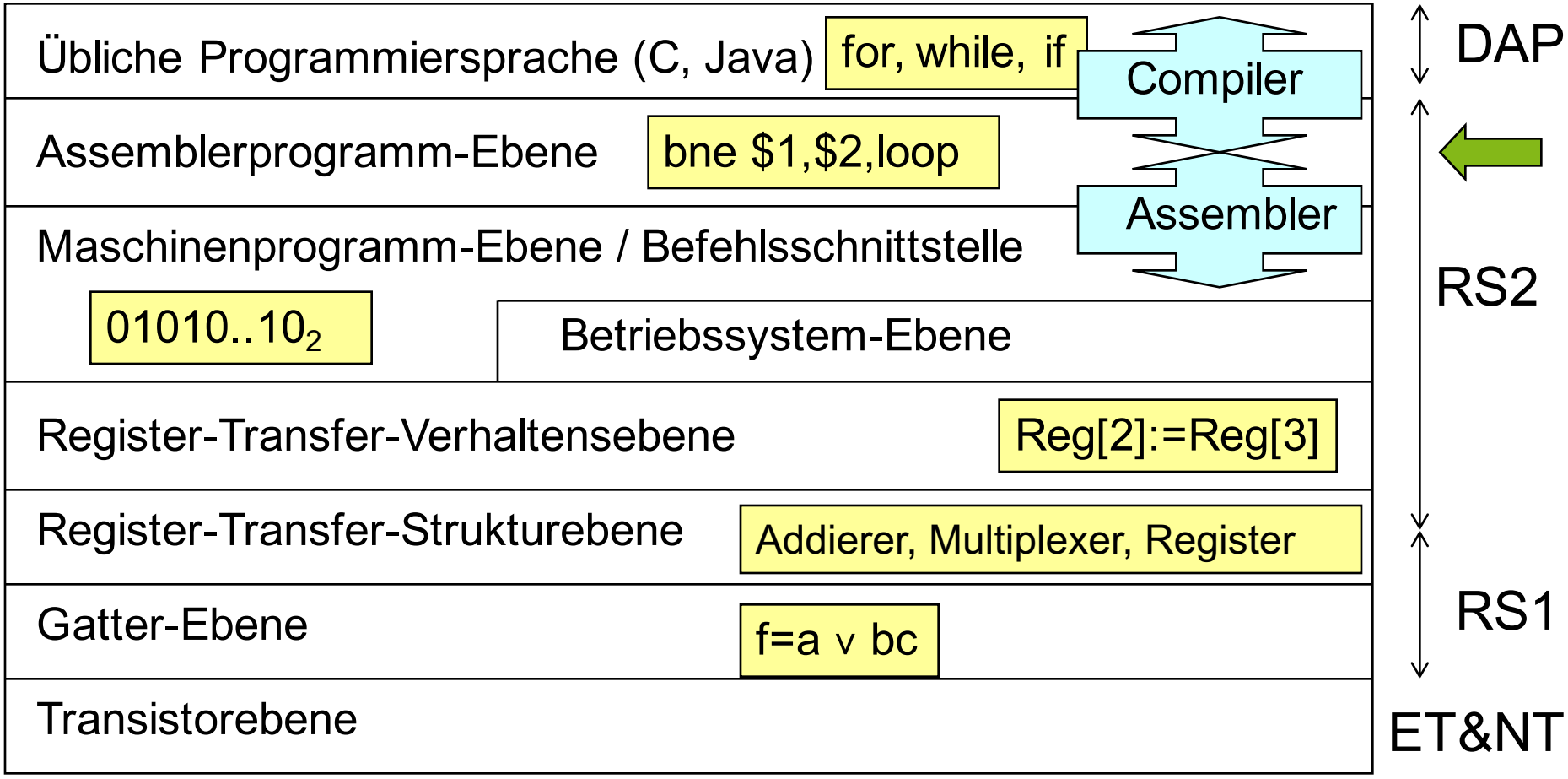
Stil:

- Betonung des Skripts
- Integration mit praktischen Übungen
  Einsatz eines Simulators

- Buch von Hennessy/Patterson: *Computer Organization: The hardware/software interface*, Morgan Kaufman, 2. Aufl. (siehe Lehrbuchsammlung) oder 3. Aufl. als grobe Leitlinie
- Weitere Bücher: siehe Literaturverzeichnis im Skript

Rechnerarchitektur
- Einleitung -

Abstraktionsebenen:



2.2 Die Befehlsschnittstelle
2.2.1 Der MIPS-Befehlssatz

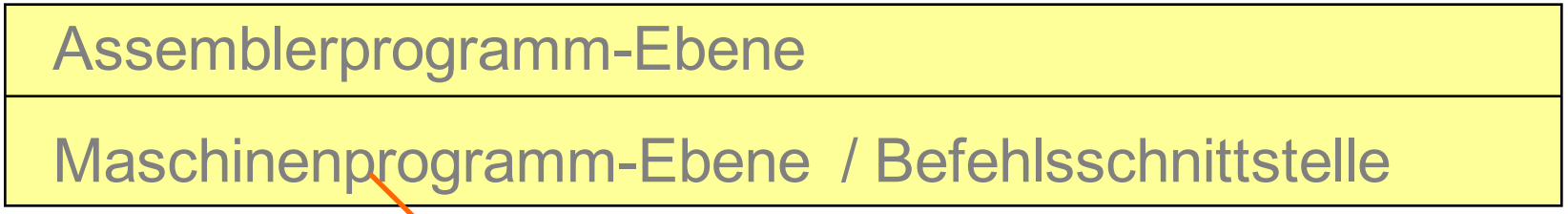
Beispiel: MIPS (~ *machine with no interlocked pipe stages*)

≠ MIPS (*million operations per second*)
Entwurf Anfang der 80er Jahre

Warum MIPS ?

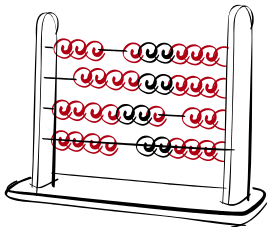
- Weitgehend sauberer und klarer Befehlssatz
- Kein historischer Ballast
- Basis der richtungsweisenden Bücher von Hennessy/Patterson
- Simulator verfügbar
- MIPS außerhalb von PCs (bei Druckern, Routern, Handys) weit verbreitet

Begriffe



- **MIPS-Befehle** sind elementare Anweisungen an die MIPS-Maschine
- Ein **MIPS-Maschinenprogramm** ist eine konkrete Folge von MIPS-Befehlen
- Die **MIPS-Maschinensprache** ist die Menge möglicher MIPS-Maschinenprogramme
- Entsprechendes gilt für die Assemblerprogramm-Ebene
- Vielfach keine Unterscheidung zwischen beiden Ebenen

MIPS-Assembler-Befehle
Arithmetische und Transportbefehle



Erstes Beispiel: Addition:

Allgemeines Format:

`add a,b,c` mit `a,b,c` ∈ `$0..$31`

`$0..$31` stehen für 32 Bit lange Register,

`a` ist das Zielregister,

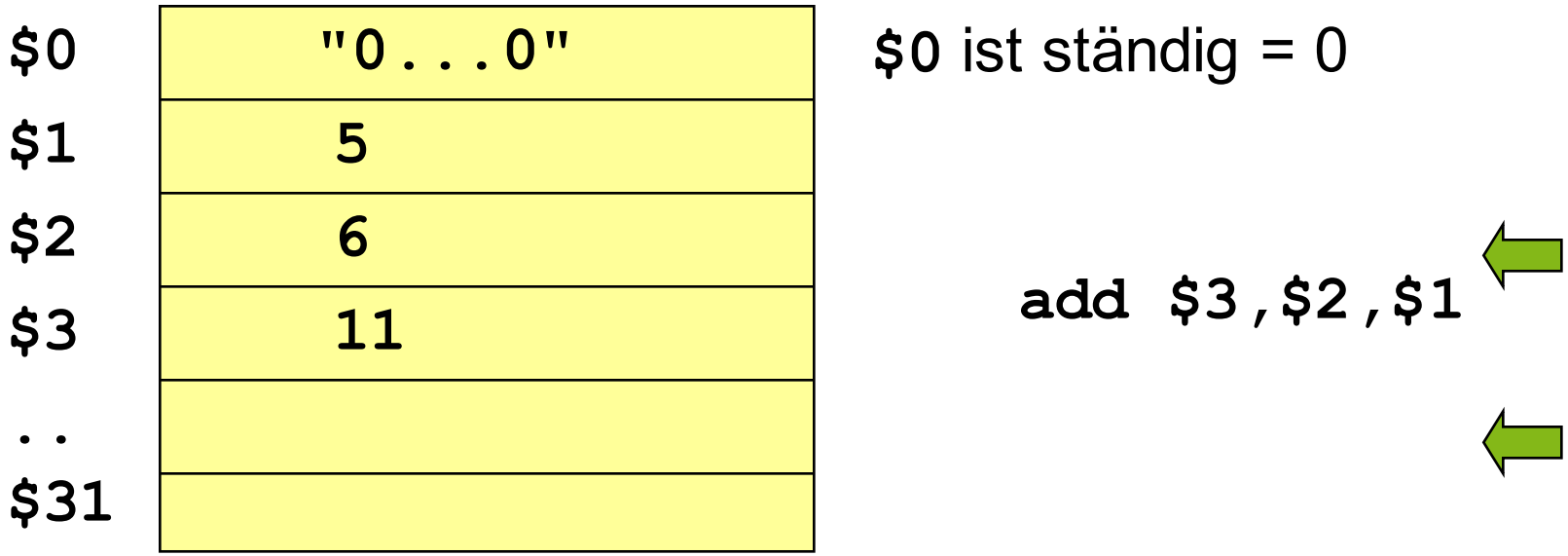
Beispiel:

```
add $3,$2,$1
```

Funktion des Additionsbefehls

Beispiel: **`add $3,$2,$1`**

Register („**Reg**“)



Simulation des MIPS-Befehlssatzes mit MARS

Simulator MARS: http://courses.missouristate.edu/kenvollmar/mars/ (erfordert Java),

Warum Simulation statt Ausführung des Intel-Befehlssatzes?

- Möglichkeit der Benutzung des MIPS-Befehlssatzes
- Keine Gefährdung des laufenden Systems
- Bessere Interaktionsmöglichkeiten
- MARS läuft auf verschiedenen Plattformen: Windows, MAC OS X, Linux

Installation jetzt dringend empfohlen, für Studierende ohne PC-Zugang Nutzung an Fakultätsrechnern möglich.

Schreiben Sie eigene kleine Programme!

Älterer Simulator: SPIM

- MARS versucht, SPIM-Obermenge zu implementieren
- Bekannte Ausnahmen:
  - .set-Anweisung
  - *Delayed branches*
  - Kein Standard-*trap handler* in MARS
  - Keine Extra-Befehlsliste verfügbar
  - Vergrößerung des Bildschirm-Fonts?
  - Programm-Ende mittels exit-syscall statt mit `jr $31`
  - Andere Realisierung großer Konstanten
- (SPIM-) Befehlsliste befindet sich im Anhang des Skripts.

Semantik: per Register-Transfer-Notation
(genauer: Register-Transfer-Verhaltens-Notation)

Argumente oder Ziele: Register oder Speicher, z.B.

```
Reg[3];  PC;  Reg[31]
```

In Anlehnung an HW-Beschreibungs-sprache VHDL

Zuweisungen: mittels `:=` , z.B.

```
PC := Reg[31]
```

Konstante Bitvektoren: Einschluss in ", z.B:

```
"01010100011"
```

Selektion von einzelnen Bits: runde Klammern:

```
PC(15:0)  -- VHDL downto statt `:´, alt: `.´ vor `(´
```

Konkatenation (Aneinanderreihung) mit `&`, z.B.

```
(Hi & Lo);
PC := PC(31:28) & I(25:0) & "00"
```

Semantik des Additionsbefehls

Bedeutung in Register-Transfer-Notation

Beispiel:

```
add $3,$2,$1        # Reg[3]:= Reg[2]+Reg[1]
```

leitet in der Assemblernotation einen Kommentar ein.

Register speichern (Teils des) aktuellen Zustands;
add-Befehl veranlasst Zustandstransformation.

Addition für 2k-Zahlen oder für vorzeichenlose Zahlen? (1)

Muss der **`add`**-Befehl „linkestes“ Bit als Vorzeichen betrachten?

Nein, bei 2k-Zahlen und bei vorzeichenlosen Zahlen („Betragszahlen“) kann jede Stelle des Ergebnisses gemäß der Gleichungen für Volladdierer bestimmt werden, unabhängig davon, ob die Bitfolgen als 2k-Zahlen oder als Betragszahlen zu interpretieren sind (siehe RS, Teil 1).

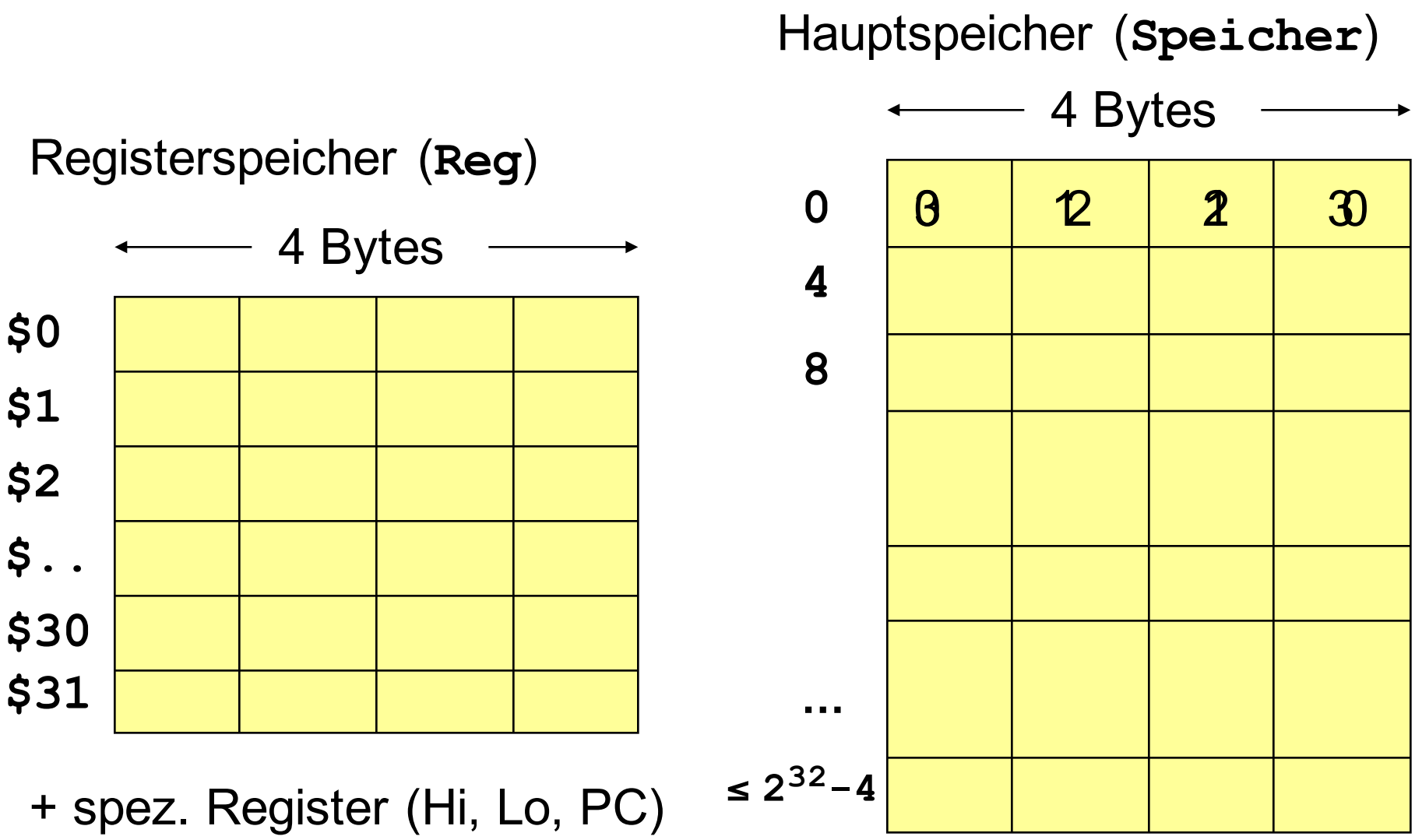
- ☞ es reicht ein Additionsbefehl zur Erzeugung der Ergebnis-Bitvektoren für beide Datentypen aus.
  Allerdings Unterschiede hinsichtlich der Behandlung von Bereichsüberschreitungen.

Addition für 2k-Zahlen oder für vorzeichenlose Zahlen? (2)

MIPS-Architektur bietet **`addu`**-Befehl für *unsigned integers:*

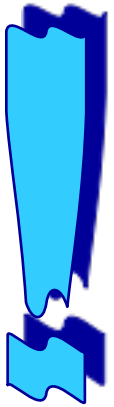
- **`add`**-Befehl signalisiert Bereichsüberschreitungen (für vorzeichenbehaftete Zahlen),
- **`addu`**-Befehl ignoriert diese (kein Signalisieren von Bereichsüberschreitungen vorzeichenloser Zahlen).

Das Speichermodell der MIPS-Architektur



Eigenschaften des Von-Neumann-Rechners (1)

Wesentliches Merkmale der heute üblichen Rechner, die auf dem Prinzip des Von-Neumann-Rechners basieren:

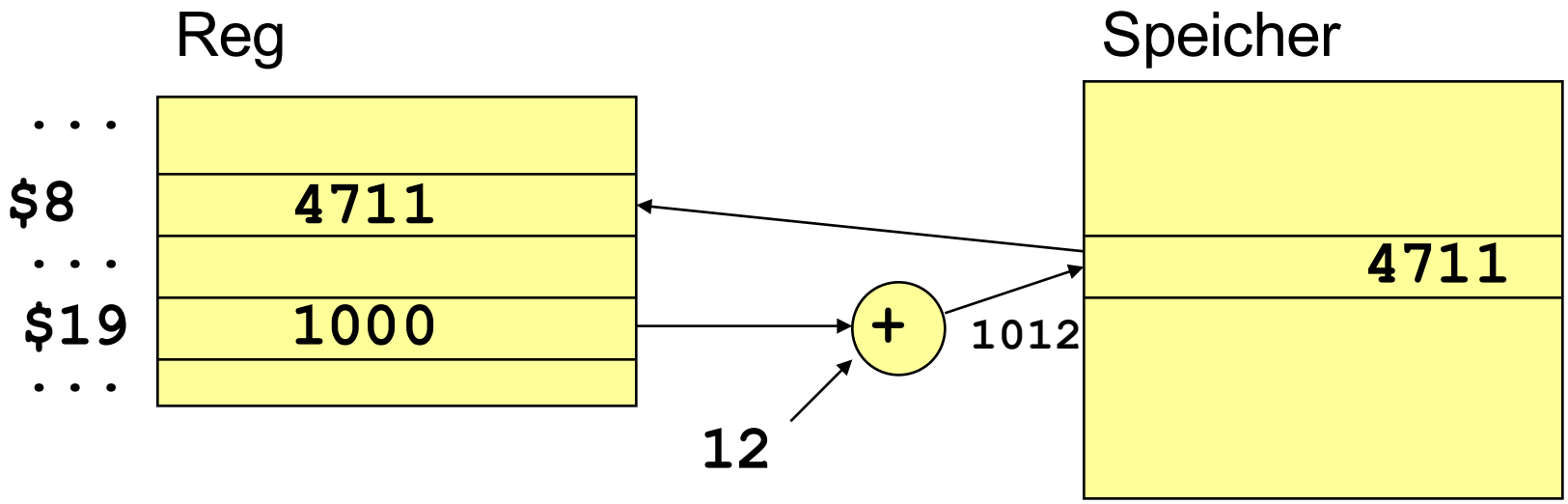


1. Einteilung des Speichers in Zellen gleicher Größe, die über **Adressen** angesprochen werden können.

Der `load word`-Befehl

Allgemeine Form:
`lw ziel, offset(reg)` mit **`ziel, reg`** ∈ **`$0..$31,`**
`offset:` Konstante ∈ -2^{15} .. $2^{15}-1$, deren Wert beim Laden des Programms bekannt sein muss bzw. deren Bezeichner.
Beispiel:

```
lw $8, 12($19)# Reg[8]:=Speicher[12+Reg[19]]
```



Der `store word`-Befehl

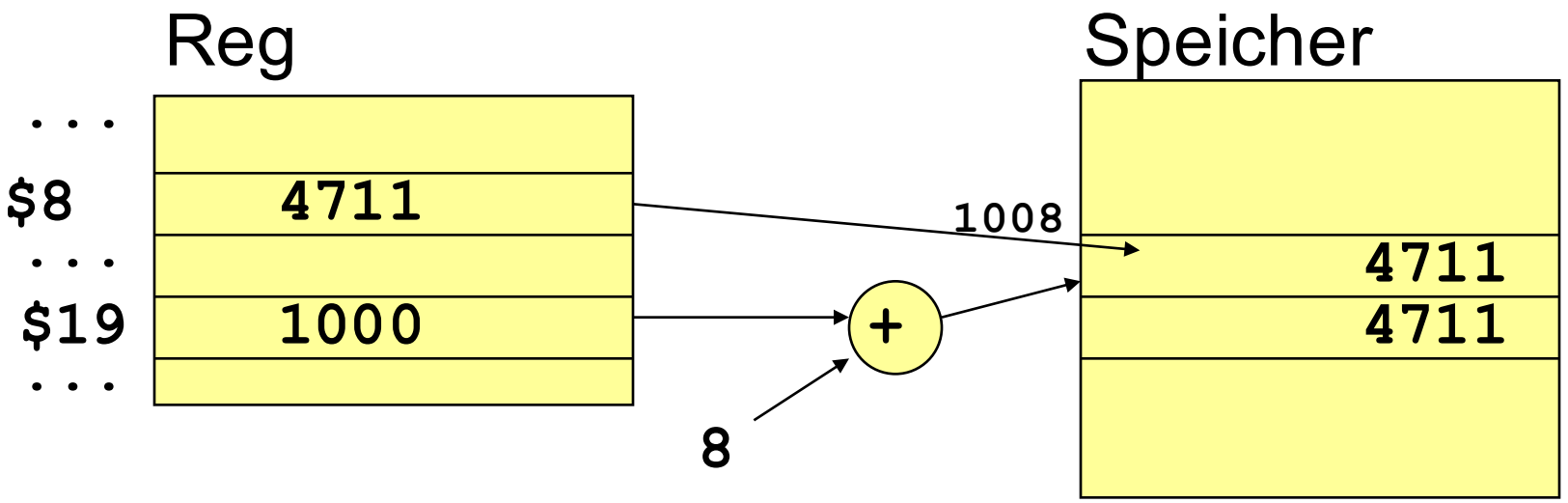
Allgemeine Form:

`sw quel, offset(reg)` mit **`quel, reg ∈ $0..$31,`**

`offset:` Konstante ∈ -2^{15} .. $2^{15}-1$, deren Wert beim Laden des Programms bekannt sein muss bzw. deren Bezeichner.

Beispiel:

`sw $8, 8($19) # Speicher[8+Reg[19]] := Reg[8]`



Darstellung von Befehlen im Rechner

Zergliederung eines Befehlswortes in **Befehlsfelder**;
Jede benutzte Zergliederung heißt **Befehlsformat;**
Beispiele:

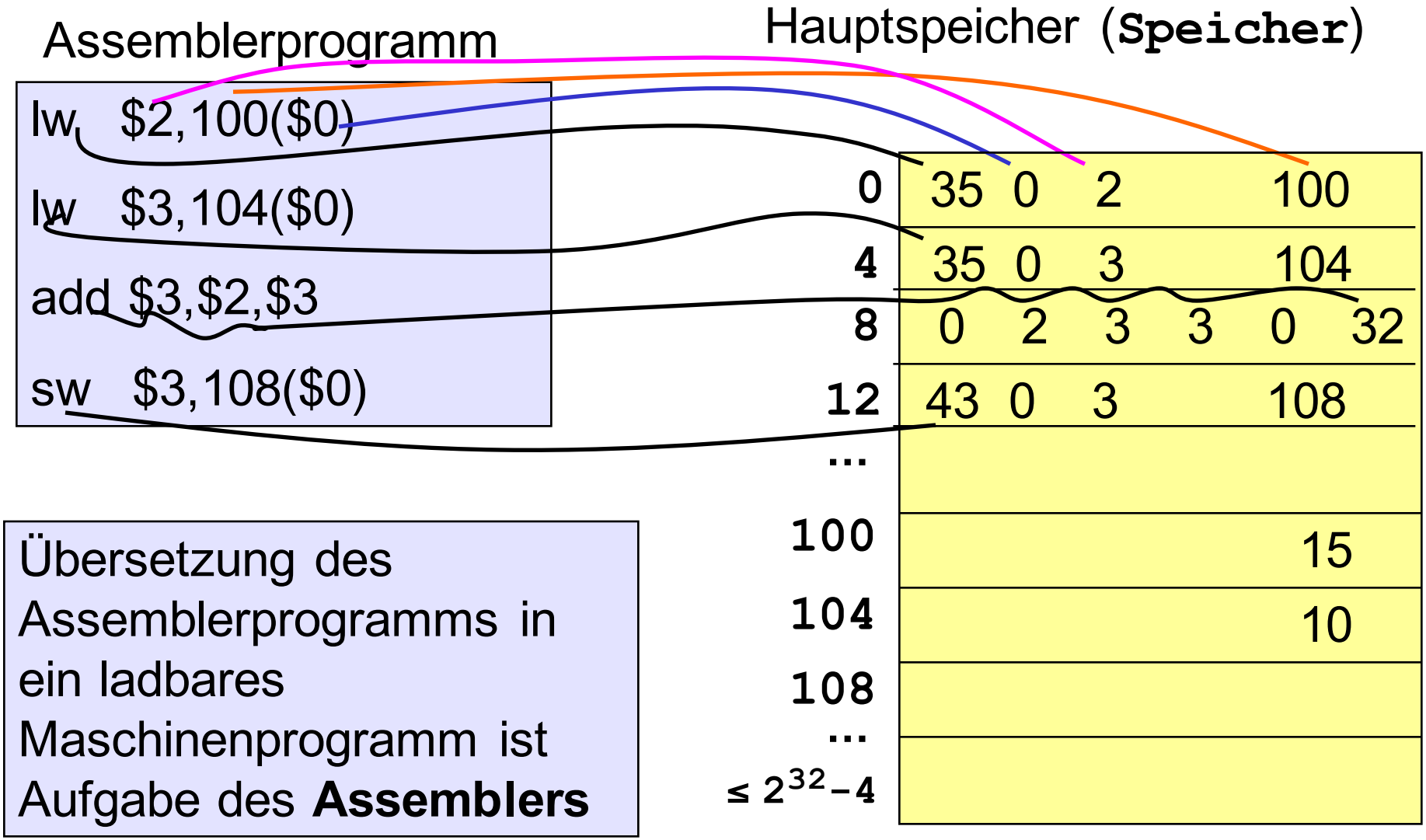
function; kodiert Befehlsvariante

	Größe [Bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
`lw, sw`	Format I	op	rs	rt	offset		

Operationscode, kodiert Befehl

shift amount; kodiert Befehlsvariante

Speicherung von Befehlen im Rechner



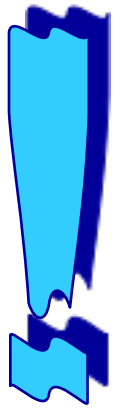
Übersetzung des Assemblerprogramms in ein ladbares Maschinenprogramm ist Aufgabe des **Assemblers**

Eigenschaften des Von-Neumann-Rechners (2)

Wesentliche Merkmale der heute üblichen Rechner, die auf dem Prinzip des Von-Neumann-Rechners basieren:

2. Verwendung von speicherprogrammierbaren Programmen.

3. Speicherung von Programmen und Daten in demselben Speicher.



Zusammenfassung

- Schichtenmodell
  - Programme in höherer Programmiersprache
  - Assemblerprogramme
  - Maschinenprogramme
  - RT-Verhalten/-Strukturen
  - Gatter
- Unterscheidung zw. Befehlen, Programmen, Sprachen
- Exemplarische Betrachtung der MIPS-Assembler- & Maschinensprache ☞ MARS
  - add-, lw-, sw-Befehle; RT-Semantik; Unterscheidung add/addu
  - Speichermodell
  - Darstellung von Befehlen
- Prinzipien der von Neumann-Maschine

Abarbeitung: immer der Reihe nach

Reg

$0	
$1	
$2	15
$3	25
..	0

Hauptspeicher (**Speicher**)

0	35 0 2 100
4	35 0 3 104
8	0 2 3 3 0 32
12	43 0 3 108
...	
100	15
104	10
108	25
...	
≤ $2^{32}-4$	

Der Zeiger auf den gerade ausgeführten Befehl wird im Programmzähler **PC** (*program counter*) gespeichert

Eigenschaften des Von-Neumann-Rechners (3)

Wesentliche Merkmale der heute üblichen Rechner, die auf dem Prinzip des Von-Neumann-Rechners basieren:

4. Die sequentielle Abarbeitung von Befehlen.

5. Es gehört zur Semantik eines jeden Befehls (Ausnahme: Sprungbefehle, s.u.), dass zusätzlich zu den erwähnten Änderungen der Speicherinhalte noch der Programmzähler PC erhöht wird, im Fall der MIPS-Maschine jeweils um 4.

2.2.1 Der MIPS-Befehlssatz
2.2.1.1 Arithmetische und Transportbefehle

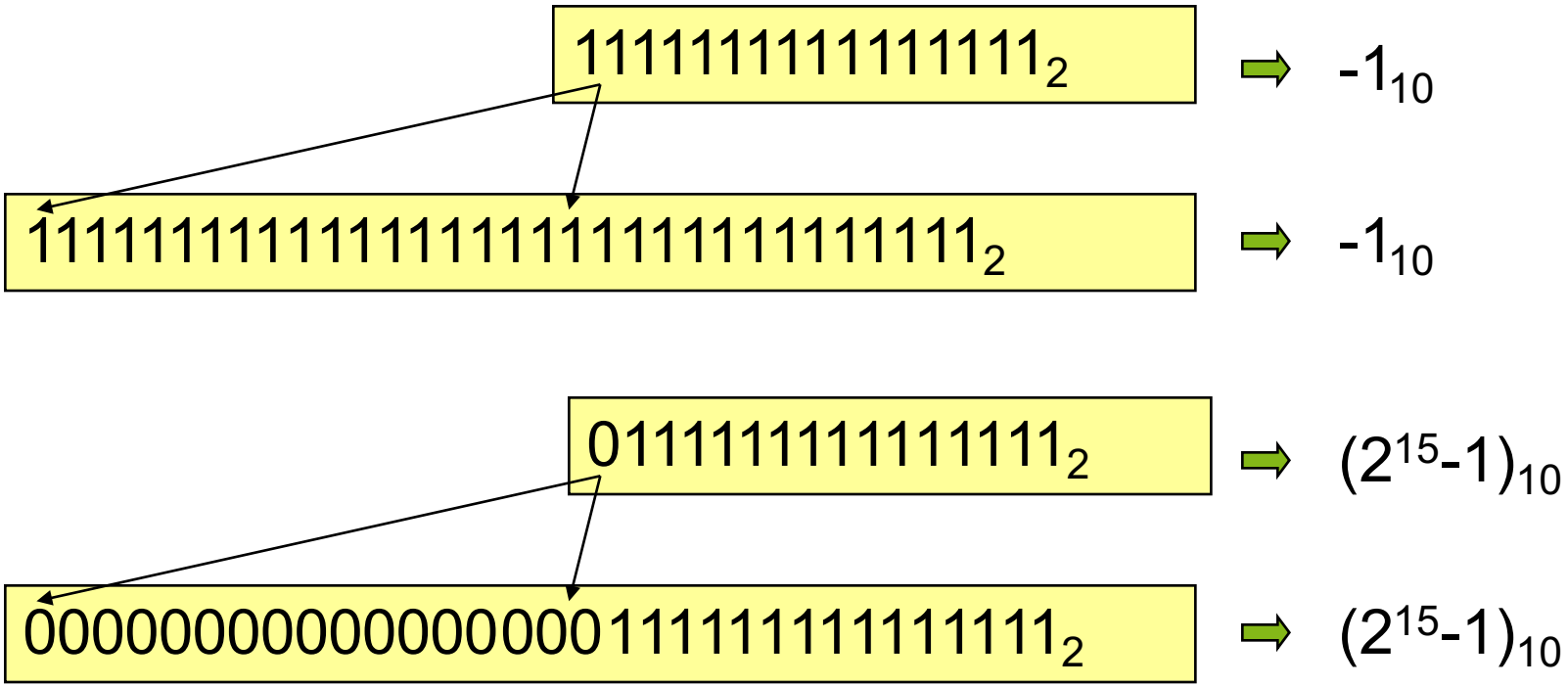
Darstellung von Befehlen im Rechner (Wdh.)

	Größe [Bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
`lw, sw`	Format I	op	rs	rt	offset		

Problem: Passt nicht zur Länge der Register

Nutzung des 16-Bit-Offsets in der 32-Bit Arithmetik

Replizierung des Vorzeichenbits liefert 32-Bit 2k-Zahl:



`sign_ext(a,m)` : Function, die Bitvektor **`a`** auf **`m`** Bits erweitert

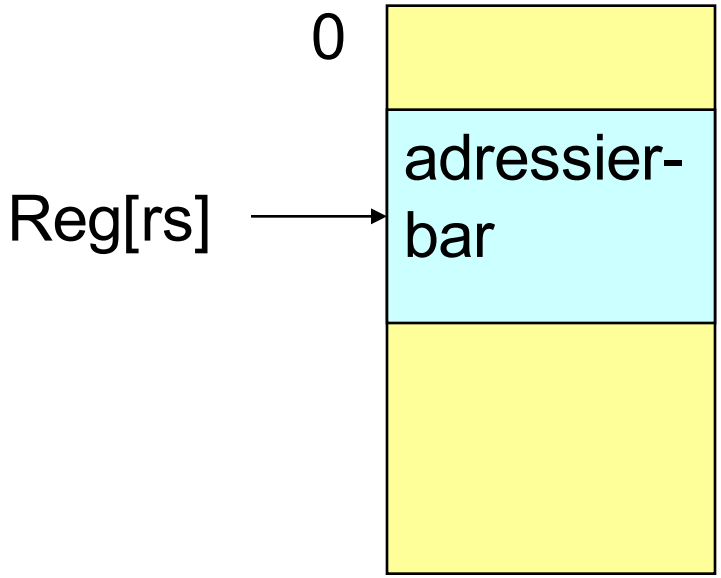
Auswirkung der Nutzung von `sign_ext` bei der Adressierung

Präzisere Beschreibung der Bedeutung des sw-Befehls:

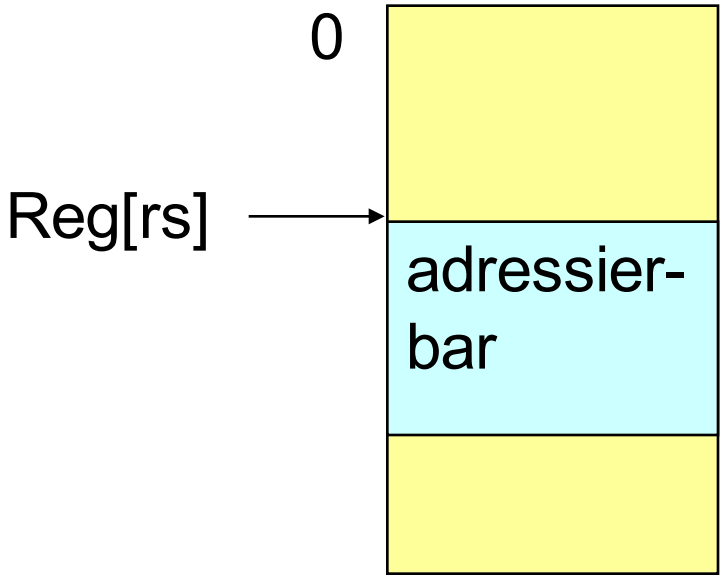
`sw rt, offset(rs) #`

`Speicher[Reg[rs]+sign_ext(offset,32)]:=Reg[rt]`

mit Vorzeichenweiterung



mit *zero-extend* (“00..00” & ...)



Einsatz der Vorzeichenerweiterung bei Direktoperanden (*immediate addressing*)

- Beschreibung der Bedeutung des *add immediate*-Befehls*

  `addi rt, rs, const #` benutzt Format I

  `# Reg[rt] := Reg[rs] + sign_ext(const,32)`

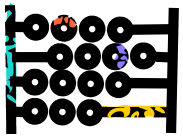
- *add immediate unsigned*-Befehl

  `addiu rt, rs, const #` benutzt Format I

  Wie **`addi`** ohne Erzeugung von Überläufen **(!)**

* Der MIPS-Assembler erzeugt vielfach automatisch *immediate*-Befehle, wenn statt eines Registers direkt eine Konstante angegeben ist, z.B. bei **`add $3,$2,3`**

Subtraktionsbefehle



```
sub $4,$3,$2 # Reg[4]:= Reg[3]-Reg[2]
```

Subtraktion, Ausnahmen möglich

```
subu $4,$3,$2 # Reg[4]:= Reg[3]-Reg[2]
```

Subtraktion, keine Ausnahmen signalisiert

Format: R

Multiplikationsbefehle

Die Multiplikation liefert doppelt lange Ergebnisse.

Beispiel: $-2^{31} \times -2^{31} = 2^{62}$;

2^{62} benötigt zu Darstellung einen 64-Bit-Vektor.

Wo soll man ein solches Ergebnis abspeichern?

MIPS-Lösung: 2 spezielle Register **`Hi`** und **`Lo:`**

```
mult $2,$3 # Hi & Lo := Reg[2] * Reg[3]
```

Höherwertiger Teil des Ergebnisses

Niederwertiger Teil des Ergebnisses

Konkatenation (Aneinanderreihung)

Transport in allgemeine Register:

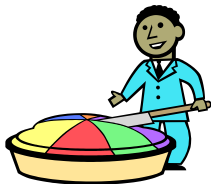
```
mfhi, $3 # Reg[3]:=Hi
mflo, $3 # Reg[3]:=Lo
```

Varianten des Multiplikationsbefehls

- **`mult $2,$3 # Hi&Lo:=Reg[2]*Reg[3] ;`**
  für ganze Zahlen in 2k-Darstellung
- **`multu $2,$3 # Hi&Lo:=Reg[2]*`$_u$ **`Reg[3] ;`**
  für natürliche Zahlen (*unsigned int)*
- **`mul $4,$3,$2`** # Besteht aus **`mult`** und **`mflo`**
  `# Hi&Lo:=Reg[3]*Reg[2]; Reg[4]:=Lo ;`
  für 2k-Zahlen, niederwertiger Teil im allgem. Reg.
- **`mulo $4,$3,$2 #`**
  `# Hi&Lo:=Reg[3]* Reg[2]; Reg[4]:=Lo ;`
  für 2k-Zahlen, niederwertiger Teil im allg. Reg, Überlauf-Test.
- **`mulou $4,$3,$2 #`**
  `# Hi&Lo:=Reg[3]*`$_u$ **`Reg[2]; Reg[4]:=Lo ;`
  für *unsigned integers*, im allg. Reg., Überlauf-Test

Merkregel:
„weniger ist mehr“
= kürzere
Bezeichung
kopiert auch in
allgem. Register

Divisionsbefehle



Problem: man möchte gern sowohl den Quotienten wie auch den Rest der Division speichern;
Passt nicht zum Konzept eines Ergebnisregisters

MIPS-Lösung: Verwendung von **`Hi`** und **`Lo`**

- **`div $2,$3 #`** für ganze Zahlen in 2k-Darstellung
  `# Lo:=Reg[2]/Reg[3]; Hi:=Reg[2] mod Reg[3]`

- **`divu $2,$3 #`** für natürliche Zahlen (*unsigned integers*)
  `# Lo:=Reg[2]/`$_u$ **`Reg[3]; Hi:=Reg[2] mod`**$_u$ **`Reg[3]`**

Skript-Anhang: "*Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run*"

Logische Befehle

Beispiel	Bedeutung	Kommentar
`and $4,$3,$2`	`Reg[4]:=Reg[3] ∧ Reg[2]`	und
`or $4,$3,$2`	`Reg[4]:=Reg[3] v Reg[2]`	oder
`andi $4,$3,100`	`Reg[4]:=Reg[3] ∧ 100`	und mit Konstanten
`sll $4,$3,10`	`Reg[4]:=Reg[3] << 10`	Schiebe nach links logisch
`srl $4,$3,10`	`Reg[4]:=Reg[3] >> 10`	Schiebe nach rechts logisch

zero_ext(

Laden von Konstanten

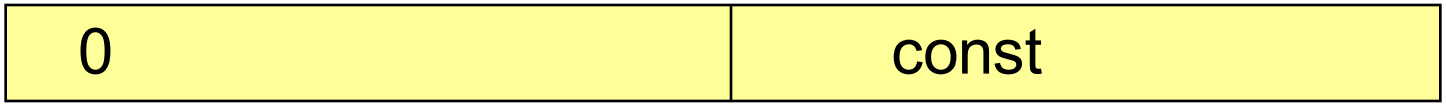
Wie kann man 32-Bit-Konstanten in Register laden?

- Direktoperanden für das untere Halbwort:

```
ori    r,s,const   # Reg[r]:=Reg[s] v (0000_16 & const)
addiu  r,s,const   # Reg[r]:=Reg[s] + zero_ext(const,32)
```

- Für den Sonderfall s=$0:

```
ori  r,$0,const    #  Reg[r]:=0 v zero_ext(const,32)
addiu r,$0,const   #  Reg[r]:=zero_ext(const,32)
```



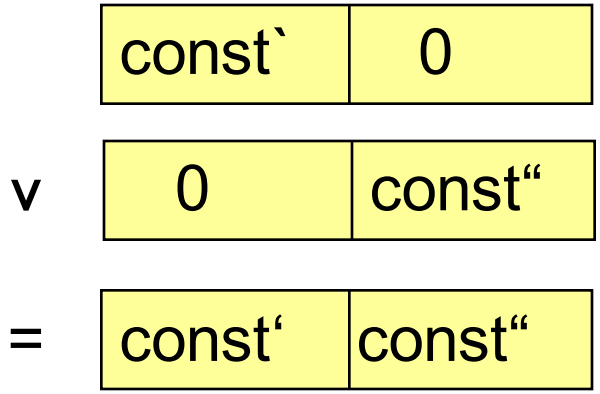
Laden von Konstanten (2)

- Für Konstanten mit: unteres Halbwort = 0
  `lui r, const` **#**Reg[r]:=const &0000_{16}
  (*load upper immediate*)

Laden von Konstanten (3)

- Für andere Konstanten:

```
lui $1, const‘
ori  r,$1,const“
```



Sequenz wird vom Assembler für **`li`** (*load immediate*) erzeugt.

Unterschiedliche Abbildung auf Maschinenbefehle bei SPIM und MARS!

Register $1 ist immer für den Assembler freizuhalten.

Der load address – Befehl `la`

In vielen Fällen muss die Adresse einer Speicherzelle in einem Register bereit gestellt werden.

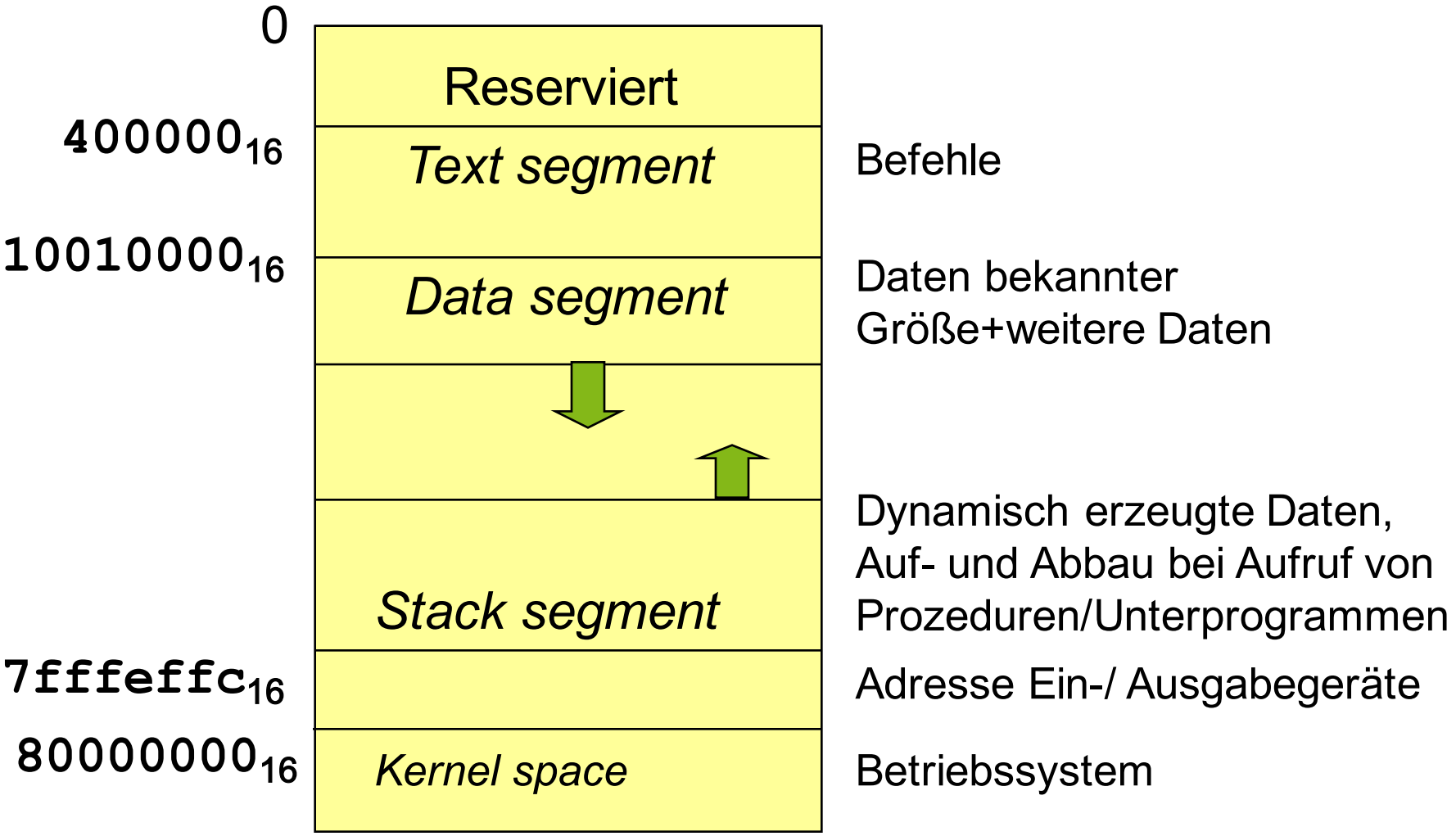
Dies ist mit den bislang vorgestellten Befehlen zwar möglich, der Lesbarkeit wegen wird ein eigener Befehl eingeführt.

Der Befehl **`la`** entspricht dem **`lw`**-Befehl,
wobei der Speicherzugriff unterbleibt. Beispiel:

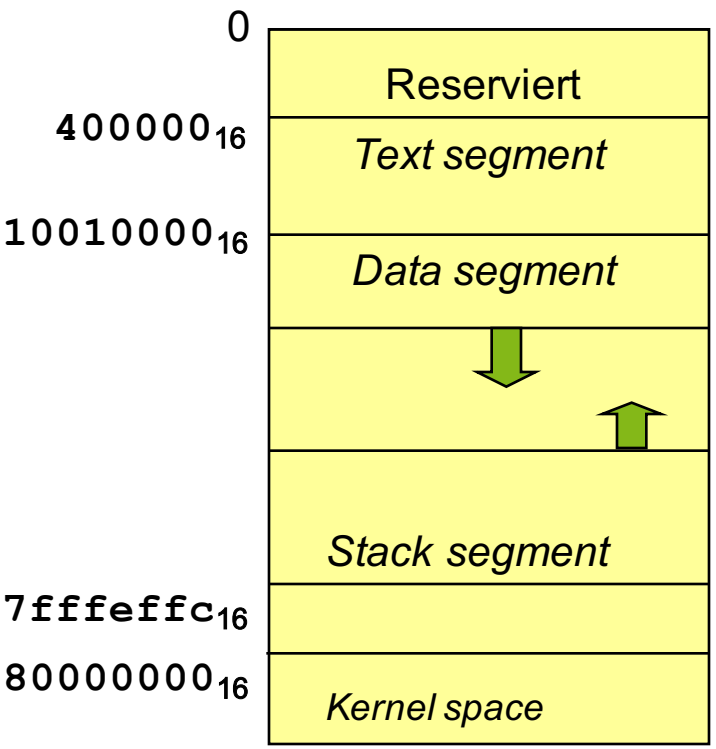
```
la $2,0x20($3) # Reg[2] := 0x20 + Reg[3]
```

`la` kann über eine Sequenz aus **`li`** und **`add`** erzeugt werden.

Einteilung des Speicherbereichs (*memory map*) im Simulator folgt üblicher C/Unix-Konvention



Problem der Nutzung nicht zusammenhängender Adressbereiche



Man müsste den Segmenten verschiedenen physikalischen Speicher zuordnen, wenn man mit diesen Adressen wirklich den physikalischen Speicher adressieren würde.

Auswege:

- Umrechnung der Adressen (siehe Abschnitt über Speicher) oder
- Benutzung weitgehend zusammenhängender Speicherbereiche

Benutzung weitgehend zusammenhängender Speicherbereiche

Betriebssystem

Text segment

Data segment

Stack segment

….

Erfordert eine relativ gute Kenntnis der Größe der Segmente

Kann beim MARS konfiguriert werden (☞ Einstellungen).

Benutzung der Speicherbereiche in einem Beispielprogramm

Beispielprogramm

```
        .glob main                  #Globales Symbol
main: lw $2, 0x10010000($0)         #Anfang Datenbereich
      lw $3, 0x10010004($0)
      add $3,$2,$3
      sw $3, 0x10010008($0)
```

Laut eig. Test erzeugt MARS im Fall „zu großer" Offsets in einem Befehl

lw $z,d($b)

Code zur Berechung der Adresse nach dem folgenden Schema in $1

lui $1,<obere 16 Bit von d>

addu $1,$1,$b; ohne *overflow check*

lw $z,<untere 16 Bit von d>($1)

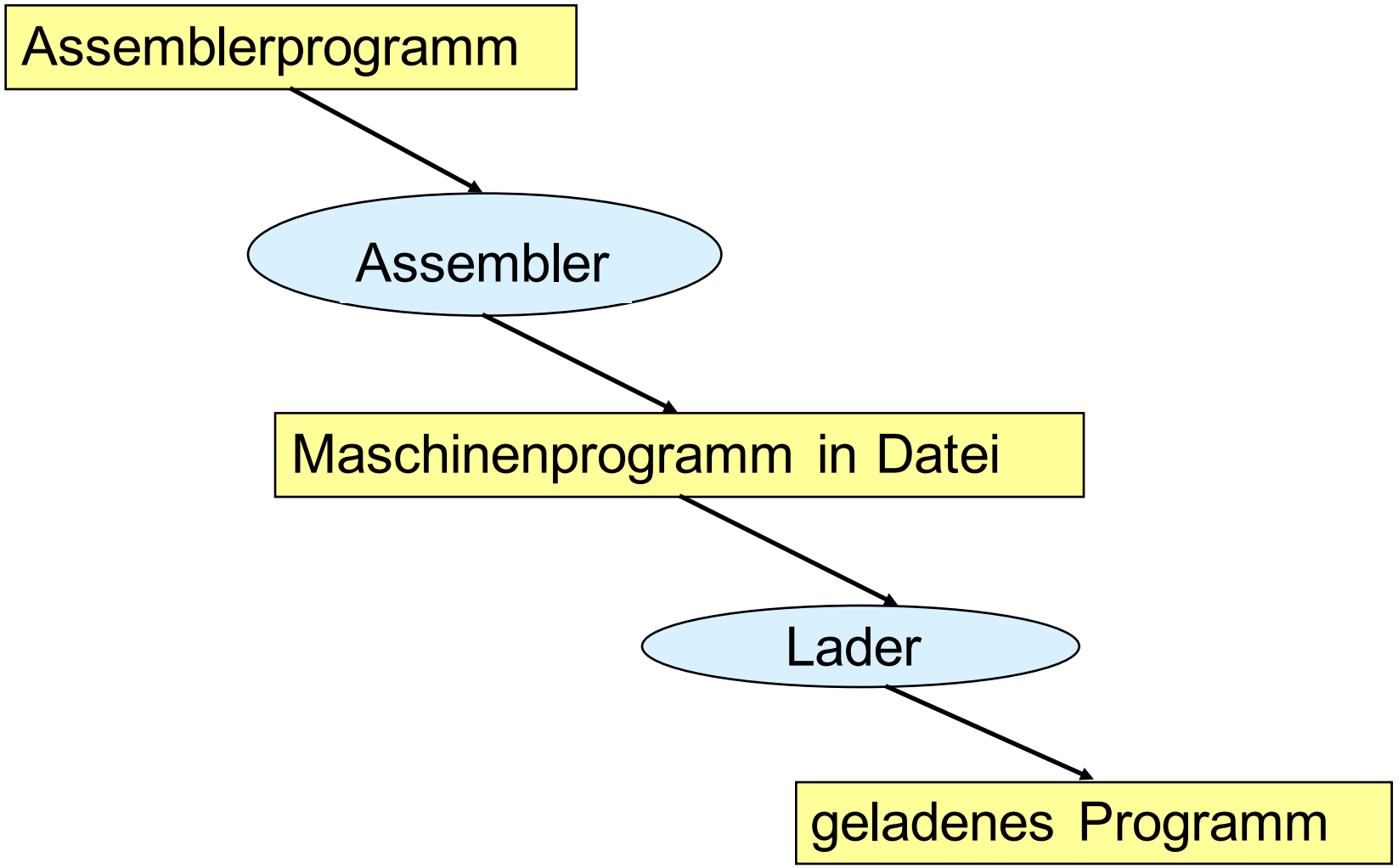
Der 2. Befehl entfällt wenn ($b) nicht vorhanden ist, aber nicht bei b=0 (!)

Zwei Versionen des Additionsprograms

```
        .globl main                    #Globales Symbol
main: lw $2, 0x10010000($0)            #Anfang Datenbereich
      lw $3, 0x10010004($0)
      add $3,$2,$3
      sw $3, 0x10010008($0)
       … syscall
```

```
        .globl main
main:  li $28,0x10010000
       lw $2, 0($28)
       lw $3, 4($28)
       add $3,$2,$3
       sw $3, 8($28)
       … syscall
```

Transformation der Programmdarstellungen



Funktion des Assemblers (1)

- Übersetzt symbolische Befehlsbezeichnungen in Bitvektoren.
- Übersetzt symbolische Registerbezeichnungen in Bitvektoren.
- Übersetzt Pseudo-Befehle in echte Maschinenbefehle.
- Verwaltet symbolische Marken.
- Nimmt die Unterteilung in verschiedene Speicherbereiche vor.

Funktion des Assemblers (2)

- Verarbeitet einige Anweisungen an den Assembler selbst:

.ascii ***text***	Text wird im Datensegment abgelegt
.asciiz ***text***	Text wird im Datensegment abgelegt, 0 am Ende
.data	die nächsten Worte sollen in das Daten-Segment
.extern	Bezug auf externes globales Symbol
.globl *id*	Bezeichner *id* soll global sichtbar sein
.kdata	die nächsten Worte kommen in das Kernel-Daten-Segment
.ktext	die nächsten Worte kommen in das Kernel-Text-Segment
.set	Setzen von Optionen (SPIM)
.space ***n***	***n*** Bytes im Daten-Segment reservieren
.text	die nächsten Worte kommen in das Text-Segment
.word *wert, … wert*	Im aktuellen Bereich zu speichern

Assembler übersetzt symbolische Registernummern

$zero = $0 usw., siehe Anhang

zero 0 Constant 0	s0 16 Saved temporary, preserved across call
at 1 Reserved for assembler	s1 17 Saved temporary, preserved across call
v0 2 Expression evaluation and	s2 18 Saved temporary, preserved across call
v1 3 results of a function	s3 19 Saved temporary, preserved across call
a0 4 Argument 1	s4 20 Saved temporary, preserved across call
a1 5 Argument 2	s5 21 Saved temporary, preserved across call
a2 6 Argument 3	s6 22 Saved temporary, preserved across call
a3 7 Argument 4	s7 23 Saved temporary, preserved across call
t0 8 Temporary, ¬ preserved across call	t8 24 Temporary, not preserved across call
t1 9 Temporary, ¬ preserved across call	t9 25 Temporary, not preserved across call
t2 10 Temporary, ¬ preserved across call	k0 26 Reserved for OS kernel
t3 11 Temporary, ¬ preserved across call	k1 27 Reserved for OS kernel
t4 12 Temporary, ¬ preserved across call	gp 28 Pointer to global area
t5 13 Temporary, ¬ preserved across call	sp 29 Stack pointer
t6 14 Temporary, ¬ preserved across call	fp 30 Frame pointer
t7 15 Temporary, ¬ preserved across call	ra 31 Return address, used by function call

Zusammenfassung

- Sequentielle Befehlsbearbeitung
- Vorzeichenerweiterung
- Laden von Konstanten
- Weitere arithmetische/logische Befehle
- Übliche Speichereinteilung
- Funktion des Assemblers

2.2.1.5 Sprungbefehle

Problem mit dem bislang erklärten Befehlssatz: keine Abfragen möglich.

Lösung: (bedingte) Sprungbefehle (*conditional branches*).

Elementare MIPS-Befehle:

`beq rega,regb,Sprungziel` (*branch if equal*)
mit **`rega, regb`** ∈ **`$0..$31`**,
und **`Sprungziel`**: 16 Bit *integer* (oder Bezeichner dafür).

`bne rega,regb,Sprungziel` (*branch if not equal*)
mit **`rega, regb`** ∈ **`$0..$31`**,
und **`Sprungziel`**: 16 Bit *integer* (oder Bezeichner dafür).

Anwendung von `beq` und `bne`

Übersetzung von

```
if (i==j) goto L1
f=g+h;
L1: f=f-i;
```

Assembler rechnet L1 in die im Maschinencode zu speichernde Konstante um.

in

```
beq $19, $20, L1          # nach L1, falls i=j
    add $16, $17, $18     # f=g+h
L1: sub $16, $16, $19     # immer ausgeführt
```

Symbolische Bezeichnung für eine Befehlsadresse.

Unbedingte Sprünge

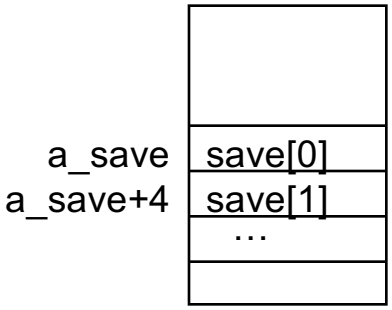
Problem: Übersetzung von

```
if (i==j) f=g+h; else f = g -h;
```

Lösung

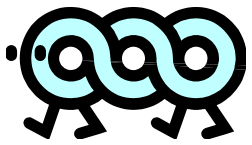
```
bne $19, $20, Else      # nach Else falls i≠j
    add $16, $17, $18   # f=g+h
    j   Exit            # Unbedingter Sprung
                        # (unconditional jump)
Else: sub $16, $17, $18 # f=g-h
Exit: ...
```

Realisierung von Array-Zugriffen



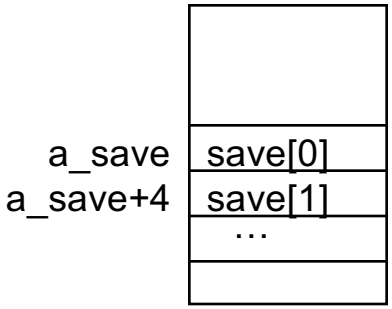
- Bei C beginnen Arrays mit dem Index 0
- Die Adresse des Arrays ist gleich der Adresse der Komponente 0 (z.B. =a_save)
- Wenn jede Array-Komponente ein Wort belegt, dann belegt Komponente mit Index i das Wort i des Arrays.
- Sei c = Anzahl der adressierbaren Speicherzellen pro Element (=4 bei 32-Bit Integern auf der MIPS-Maschine),
  sei a_save = Array-Anfangsadresse,
  → (a_save+ i x c) ist Adresse der Komponente i.

Realisierung von Schleifen mit unbedingten Sprüngen



Problem: Übersetzung von

```
while (save[i]==k) i = i+j;
```

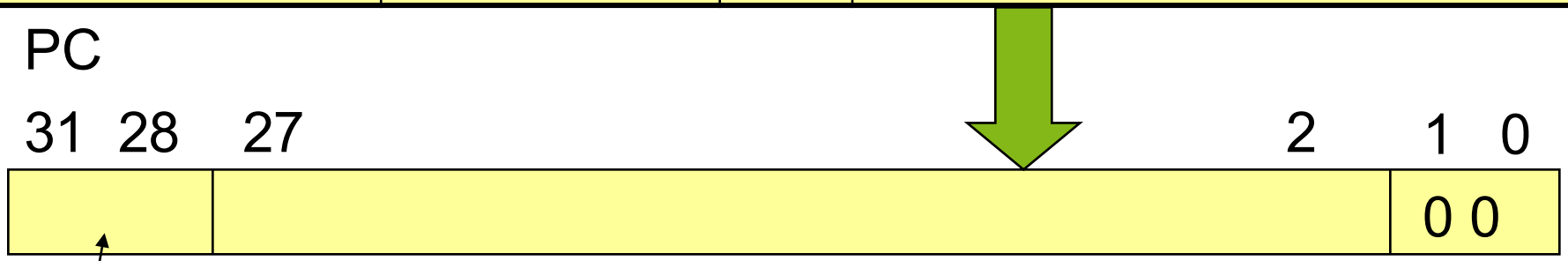


Lösung:

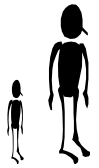
```
        li   $10,4            # Reg[10]:=4
Loop:   mul  $9,$19,$10       # Reg[9]:=i * 4
        lw   $8,a_save($9)    # Reg[8]:=save[i]
        bne  $8,$21, Exit     # nach Exit, falls ≠
        add  $19,$19,$20      # i=i+j
        j    Loop
Exit:   ...
```

Format und Bedeutung unbedingter Sprünge

	Größe [Bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
lw, sw, beq, bne	Format I	op	rs	rt	offset		
j, jal	Format J	op	adresse				



Tests auf <, ≤ ,>, ≥



MIPS-Lösung: **`slt`**-Befehl (*set if less than*)

```
slt ra,rb,rc  #
# Reg[ra]:=if Reg[rb]<Reg[rc] then 1 else 0
```

Tests werden vom Assembler aus **`slt, bne`** und **`beq`**-Befehlen zusammengesetzt:

Beispiel:
aus **`blt $2,$3,L`** wird

```
slt $1,$2,$3
bne $1,$0,L
```

`$1` ist per Konvention dem Assembler vorbehalten

Weitere Befehle

Weitere Befehle:

- slti (Vergleich mit Direktoperanden)
- sltu (Vergleich für Betragszahlen)
- sltui (Vergleich für Betragszahlen als Direktoperand)
- ble (Verzweige für *less or equal*)
- blt (Verzweige für *less than*)
- bgt (Verzweige für *greater than*)
- bge (Verzweige für *greater or equal*)

} Pseudo-befehle

Übersetzung einer `for`-Schleife

Bedingte Sprünge werden auch zur Übersetzung von **`for`**-Schleifen benötigt. Beispiel: Das C-Programm

```
j = 0;
for (i=0; i<n; i++)   j = j+i;
```

kann in das folgende Programm übersetzt werden:

```
main:   li    $2,0                  # j:=0
        li    $3,0                  # i:=0
loop:   bge   $3,$4, ende           # i>=n?
        add   $2,$2,$3              # j:=j+i
        addi  $3,$3,1               # i:=i+1
        j     loop
ende:   ...
```

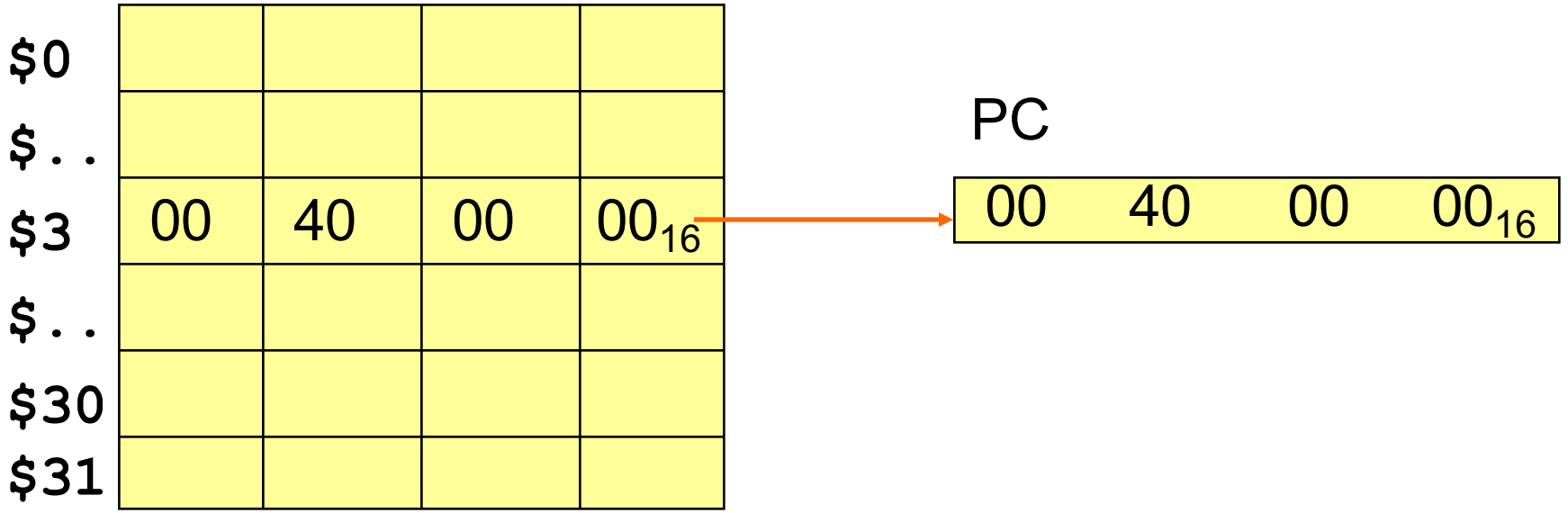
Realisierung von berechneten Sprüngen: der `jr`-Befehl

Format: **`jr reg`** (jump register), mit **`reg ∈ $0..$31`**

Semantik: **`PC:=Reg[reg]`**

Beispiel: **`jr $3`**

Registerspeicher (**`Reg`**)



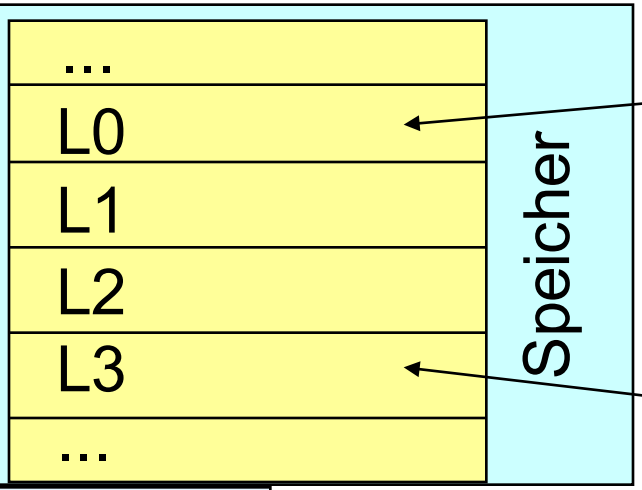
SWITCH-Anweisungen

Annahme: k=2

```
switch(k)  { /* k muss ∈ 0..3 sein! */
 case 0: f=i+j; break; /* k =0 */
 case 1: f=g+h; break; /* k =1 */
case 2: f=g-h; break; /* k =2 */
 case 3: f=i-j; break; /* k =3 */
}
```

Realisierung

Jumptable



Adresse des ersten Befehls für case 0

Adresse des ersten Befehls für case 3

```
.data
Jumptable:  .word L0,L1,L2,L3, ..
```

Realisierung von SWITCH-Anweisungen mittels des jr-Befehls

```
switch(k) {
 case 0: f=i+j; break; /* k =0 */
 case 1: f=g+h; break; /* k =1 */
 case 2: f=g-h; break; /* k =2 */
 case 3: f=i-j; break; /* k =3 */ }
```

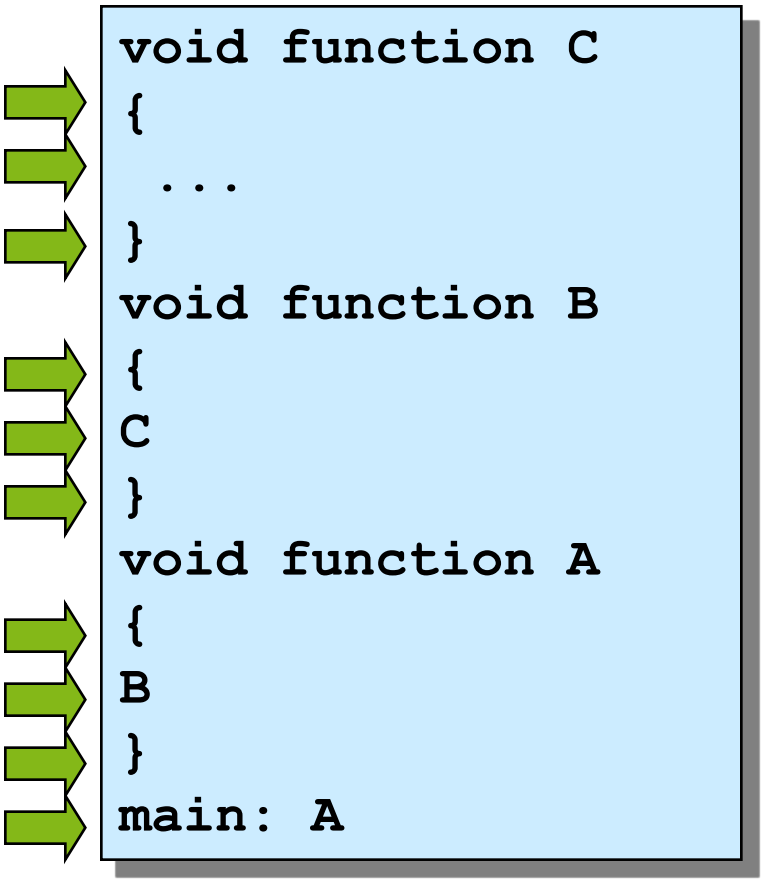
```
      li   $10,4              # Reg[10]:=4
swit: mul  $9,$10,$21         # Reg[9]:=k*4
      lw   $8,Jumptable($9)   # 1 der 4 Adressen
      jr   $8                 # PC:=Reg[8]
L0:   add  $16,$19,$20        # k ist 0;
      j    Exit               # entspricht break
L1:   add  $16,$17,$18        # k ist 1;
      j    Exit               # entspricht break
L2:   sub  $16,$17,$18        # k ist 2;
      j    Exit               # entspricht break
L3:   sub  $16,$19,$20        # k ist 3;
Exit: ...                     #
```

Zusammenfassung

- Sprungbefehle
  - Bedingte
  - Unbedingte Sprungbefehle

- Realisierung von
  - if-Anweisungen
  - while- und for-Schleifen
  - switch-Anweisungen

2.2.1.6 Prozeduraufrufe



```
void function C
{
  ...
}
void function B
{
C
}
void function A
{
B
}
main: A
```

Man muss

- sich merken können, welches der auf den aktuellen Befehl im Speicher folgende ist (d.h., man muss sich PC+4 merken) und
- an eine (Befehls-) Adresse springen.

Der *jump-and-link*-Befehl

Format: **`jal adresse`**

wobei **`adresse`** im aktuellen 256 MB-Block liegt

Bedeutung:

```
Reg[31]:=PC+4;
PC:=PC(31:28) & I(25:0) & "00";
    (auch PC:=(0xf0000000 & PC)|addresse << 2);
```

Beispiel: **`jal 0x104000`** **`# addresse << 2 ist 0x410000`**

Registerspeicher (**`Reg`**)

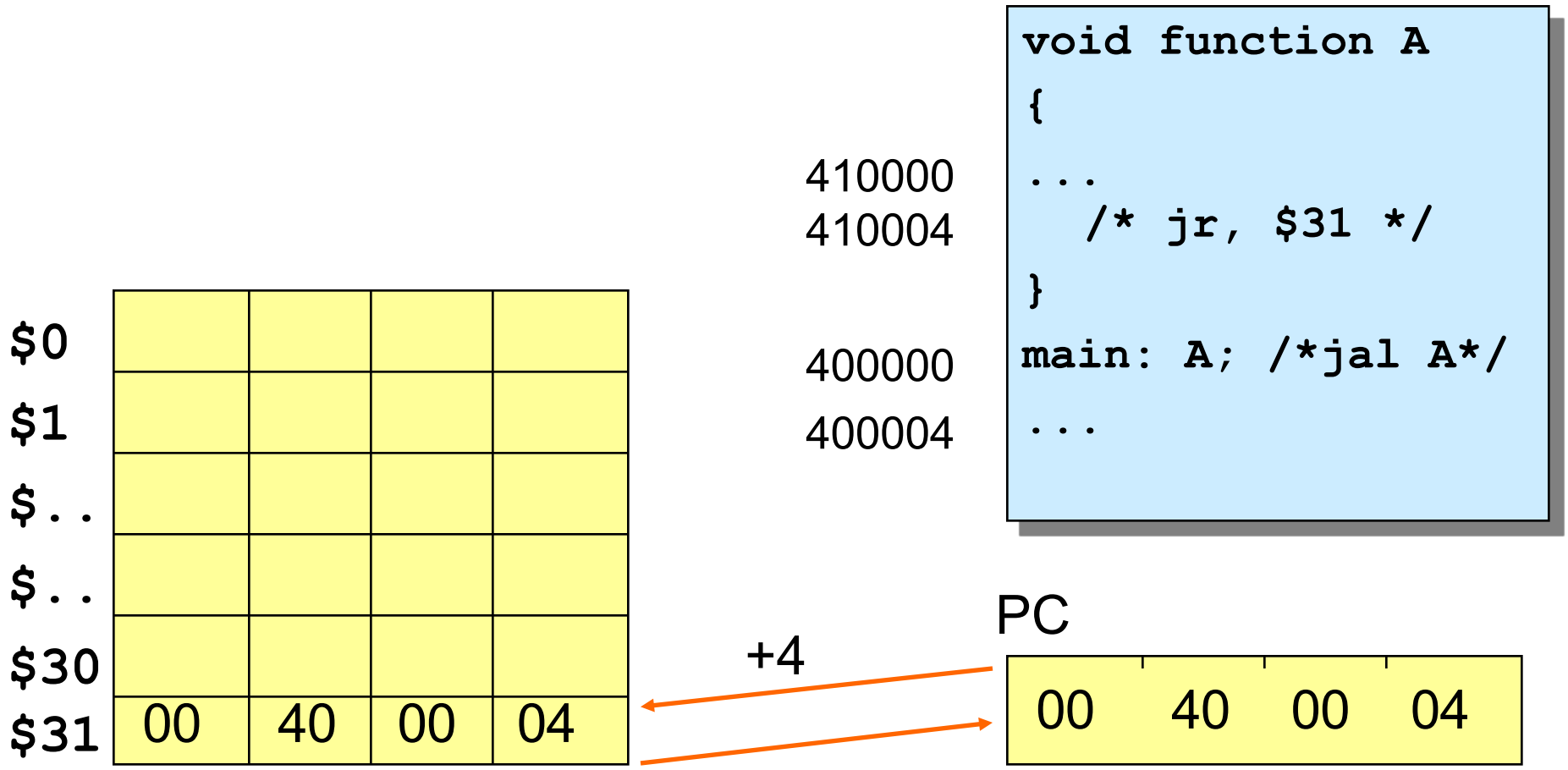
`$0`				
`$1`				
`$..`				
`$..`				
`$30`				
`$31`	00	40	00	04_{16}

Nach der Ausführung von **jal** an Adresse 400000_{16}

PC

00	41	00	00_{16}

Realisierung nicht verschachtelter Prozeduraufrufe mit `jal` und `jr`

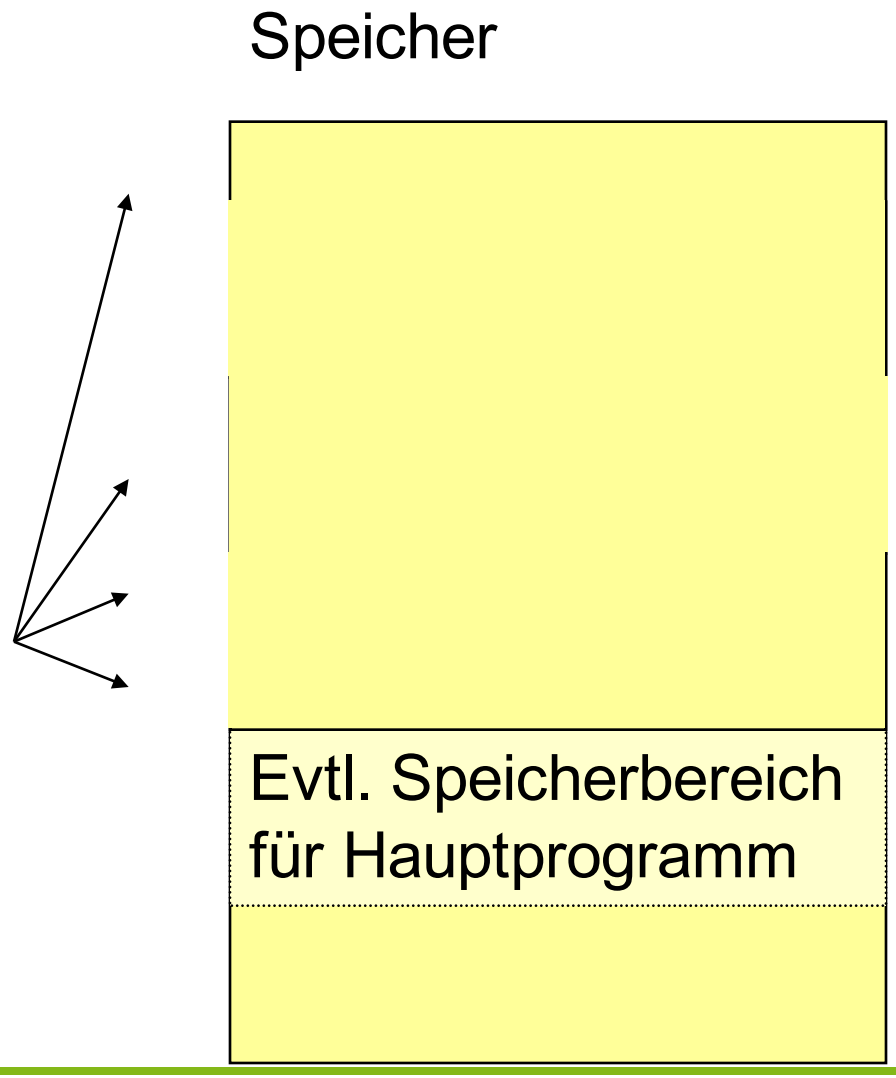


Das Stapel-Prinzip

```
void function C
{
  ...
}
void function B
{
C
}
void function A
{
B
}
main: A
```

Evtl. Speicherbereich für Hauptprogramm

Realisierung eines Stapels im Speicher



MIPS-Konvention: Benutzung von Register 29 als *stack-pointer*

Konkrete Realisierung für drei Prozeduren

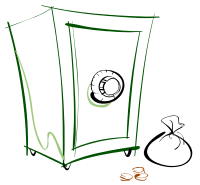
Statisch:

```
C()
  { ...; }    /* jr*/
B()
  {          /*$31->stack*/
   C();
  }          /*stack->$31,jr */
A()
  {          /*$31->stack*/
   B();
  }         /*stack->$31,jr */
main()
 {A();}
```

Dynamisch:

Aktiv	Befehl
main	jal A
A	$31 -> stack
A	jal B
B	$31 -> stack
B	jal C
C	jr $31
B	Rückschreiben von $31
B	jr $31
A	Rückschreiben von $31
A	jr $31
main	

Sichern von Registerinhalten



Einige Register sollten von allen Prozeduren genutzt werden können, unabhängig davon, welche Prozeduren sie rufen und von welchen Prozeduren sie gerufen werden.

Die Registerinhalte müssen beim Prozeduraufruf gerettet und nach dem Aufruf zurückgeschrieben werden.

2 Methoden:

1. Aufrufende Prozedur rettet vor Unterprogrammaufruf Registerinhalte (*caller save*) und kopiert sie danach zurück.
2. Gerufene Prozedur rettet nach dem Unterprogramm-Aufruf diese Registerinhalte und kopiert sie vor dem Rücksprung zurück (*callee save*).

Sichern von Registern (2)

caller save

```
caller: Retten der Register  auf
        den stack.
        jal ...
callee:
        Retten von $31.
        Befehle f. Rumpf.
        Rückschreiben von $31.
        jr $31.
caller:
        Rückschreiben der
        Register.
```

callee save

```
caller: jal ...
callee:
    Retten der Register  auf
    den stack.
    Retten von $31.
    Befehle f. Rumpf.
    Rückschreiben von $31.
    Rückschreiben der
    Register.
    jr $31.
caller:
```

2.2.1.7 Prozeduren mit Parametern

```
int stund2sec(int stund)
{return stund*60*60}
stund2sec(5);
```

Wo findet stund2sec den Eingabeparameter?

Konflikt:

- Parameter möglichst in Registern übergeben (schnell)
- Man muss eine beliebige Anzahl von Parametern erlauben.

MIPS-Konvention:
Die ersten 4 Parameter werden in Registern $4, $5, $6, $7 übergeben, alle weiteren im *stack*.

Benutzung der Register

Register	Verwendung
$0	0
$1	Assembler
$2,$3	Funktionsergebnis
$4-$7	Parameter
$8-$15	Hilfsvariable, nicht gesichert
$16-$23	Hilfsvariable, gesichert
$24-$25	Hilfsvariable, nicht gesichert
$26-$27	Für Betriebssystem reserviert
$28	Zeiger auf globalen Bereich
$29	*Stack pointer*
$30	Zeiger auf Datenbereich
$31	Rückkehradresse

Call by value vs. *call by reference*

An Prozeduren Parameterwert oder dessen Adresse übergeben?

Parameterwert selbst (***call by value***):

Gerufener Prozedur ist nur der Wert bekannt.

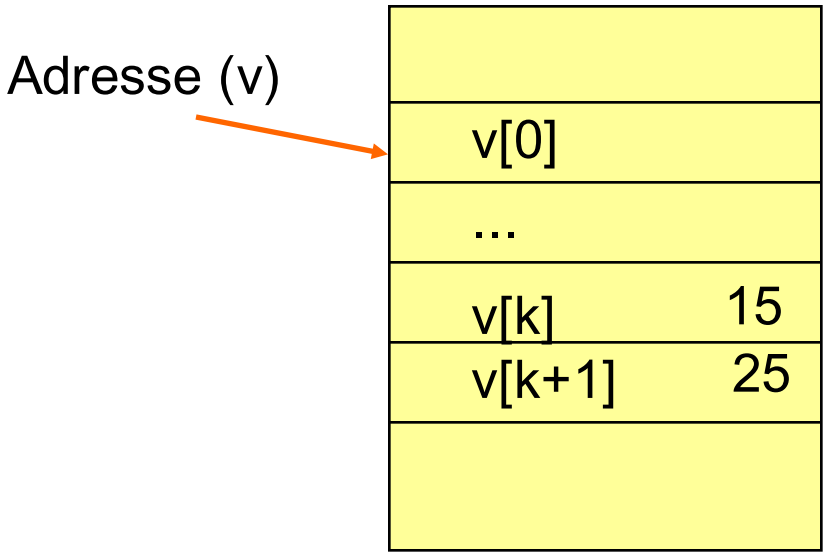
Die Speicherstelle, an der er gespeichert ist, kann nicht verändert werden.

Adresse des Parameters (***call by reference***):
Erlaubt Ausgabeparameter, Änderung der Werte im Speicher durch die gerufene Prozedur möglich.
Bei großen Strukturen (*arrays*) effizient.

C: - Bei skalaren Datentypen (*int*, Zeigern, usw.): wählbar, typischerweise *call by value*,
- bei komplexen Datentypen (*arrays*): *call by reference*.

Prozedur swap , Prinzip der Übersetzung in Assembler

```
swap (int v[], int k)
{int temp;
temp=v[k];
v[k]=v[k+1];
v[k+1]=temp;
}
```



Schritte der Übersetzung in Assembler:

1. Zuordnung von Speicherzellen zu Variablen
2. Übersetzung des Prozedurrumpfes in Assembler
3. Sichern und Rückschreiben der Register

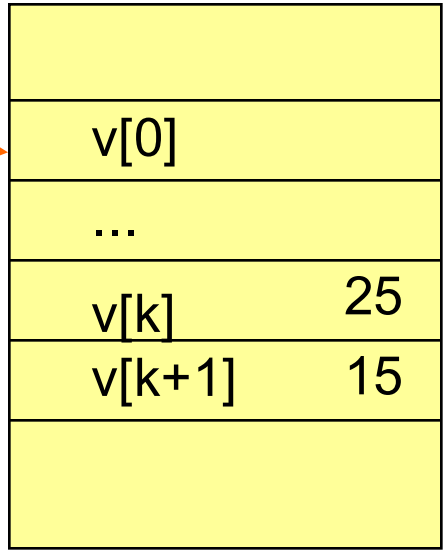
Zuordnung von Registern

Variable	Speicherzelle	Kommentar
Adresse von v	$4	Aufgrund der MIPS-Konvention für den 1. Parameter
k	$5	Aufgrund der MIPS-Konvention für den 2. Parameter
temp	$15	(irgendein freies Register)
Interne Hilfsvariable	$16	(irgendein freies Register)

Realisierung des Rumpfes

```
temp=v[k];
v[k]=v[k+1];
v[k+1]=temp;
```

$4=
Adresse (v)



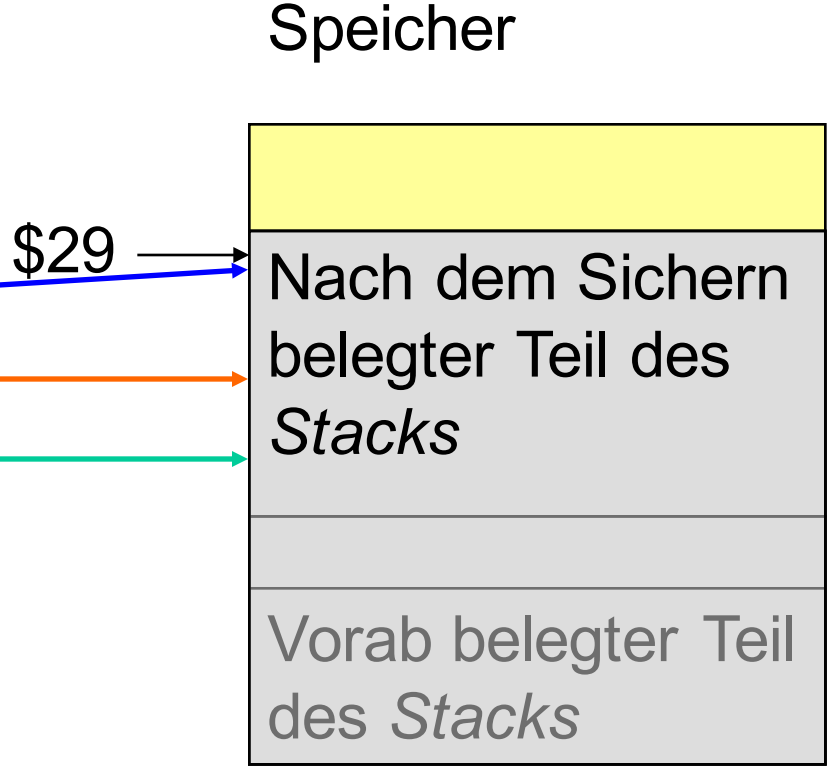

```
li   $2, 4          #
mul  $2,$2,$5       #Reg[2]:=4*k
add  $2,$4,$2       #Adresse von v[k]
lw   $15, 0($2)     #lade v[k]
lw   $16, 4($2)     #lade v[k+1]
sw   $16, 0($2)     #v[k]:=v[k+1]
sw   $15, 4($2)     #v[k+1]:=temp
```

Sichern der Register

Prolog: Sichern der in swap überschriebenen Register:

- Prolog ⇐
- Prozedurrumpf
- Epilog

```
addi $29,$29,-16
sw $2, 0($29)
sw $15, 4($29)
sw $16, 8($29)
sw $31, 12($29)
```

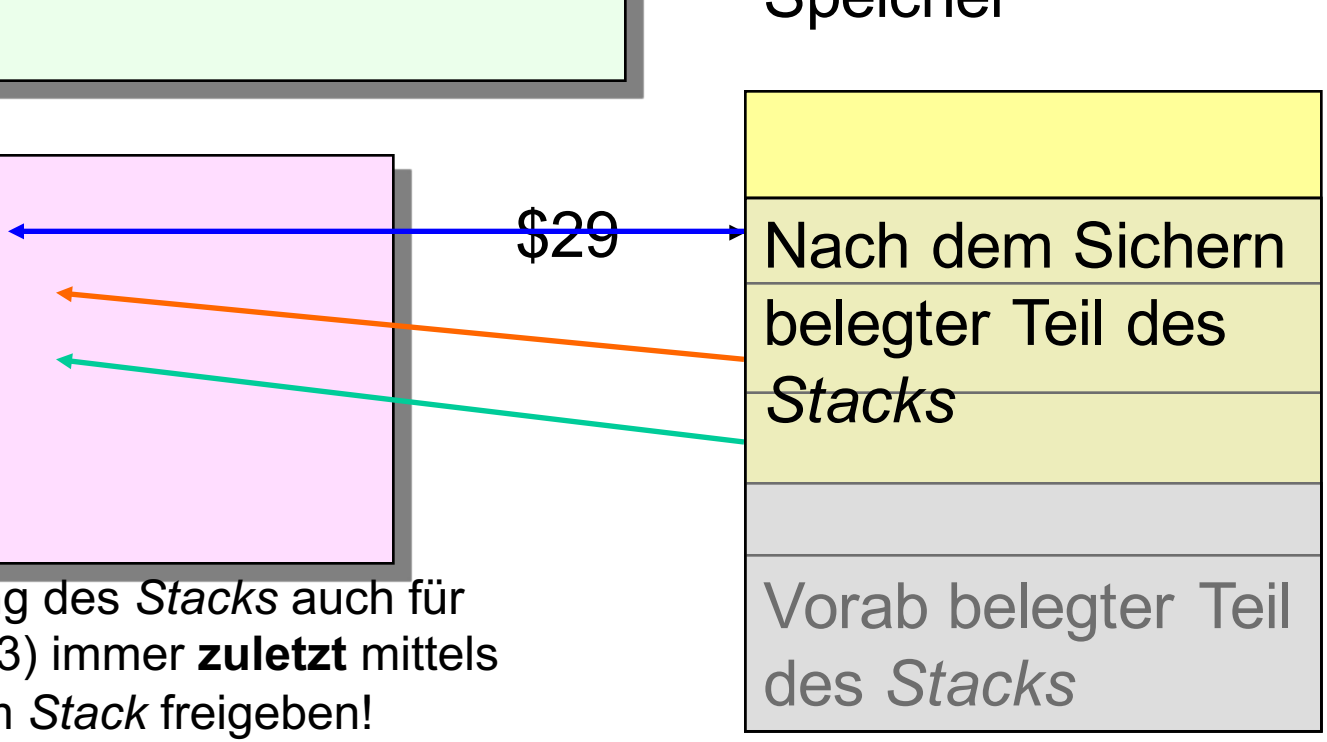


Wegen der Verwendung des *Stacks* auch für *Interrupts* (siehe 2.2.2.3) immer **zuerst** mittels `addi` den Platz auf dem *Stack* schaffen!

Rückschreiben der Register

Epilog: Rückschreiben der in swap überschriebenen Register:

- Prolog
- Prozedurrumpf
- Epilog



Wegen der Verwendung des *Stacks* auch für *Interrupts* (siehe 2.2.2.3) immer **zuletzt** mittels **addi** den Platz auf dem *Stack* freigeben!

Übersetzung von *bubble sort*

```
int v[10000]
sort(int v[], int n)
{int i,j;
  for (i=0; i<n; i=i+1) {
   for (j=i-1; j>=0 && v[j]>v[j+1]; j=j-1){
     swap(v,j);
   }
  }
}
```

Schritte der Übersetzung in Assembler:

1. Zuordnung von Speicherzellen zu Variablen
2. Übersetzung des Prozedurrumpfes in Assembler
3. Sichern und Rückschreiben der Registerinhalte

Zuordnung von Registern

Adresse von v	$4	Lt. Konvention
n	$5	Lt. Konvention
j	$17	gesichertes Register
Kopie von $4	$18	Sichern der Parameter zur Vorbereitung des Aufrufs von swap
i	$19	gesichertes Register
Kopie von $5	$20	wie $4
Hilfsvariable	$15,$16,$24,$25	irgendwelche freien Register
Hilfsvariable	$8	nicht gesichert

Übersetzung des Rumpfes

```
int v[10000]
sort(int v[], int n)
{int i,j;
 for(i=0;i<n;i=i+1){
  for(j=i-1;j>=0 &&
   v[j]>v[j+1];j=j-1)
   {swap(v,j);}}}
```

Adr. von v	$4
n	$5
j	$17
Kopie von $4	$18
i	$19
Kopie von $5	$20
Hilfsvariable	$15,$16,$24,$25
Hilfsvariable, nicht gesichert	$8

```
      add $18,$4,$0    #
      add $20,$5,$0    #
      li $19,0         # i:=0
for1: bge $19,$20,ex1  # i>=n?
      addi $17,$19,-1  # j:=i-1
for2: slti $8,$17,0    # j<0?
      bne $8,$0,ex2    #
      li $8,4          # $8:=4
      mul $15,$17,$8   # j*4
      add $16,$18,$15  # Adr(V[j]
      lw $24,0($16)    # v[j]
      lw $25, 4($16)   # v[j+1]
      ble $24,$25,ex2  # v[j]<=
      add $4,$18,$0    # 1.Param.
      add $5,$17,$0    # 2.Param.
      jal swap
      addi $17,$17,-1  # j:=j-1
      j for2           # for-ende
ex2:  addi $19,$19,1   # i:=i+1
      j for1           # for-ende
ex1 : # ende
```

Sichern der Registerinhalte

Prolog: Sichern der über-schriebenen Register:

Prolog

Prozedurrumpf

Epilog

Adr. von v	$4
n	$5
j	$17
Kopie von $4	$18
i	$19
Kopie von $5	$20
Hilfsvariable	$15,$16,$24,$25
Hilfsvariable, nicht gesichert	$8

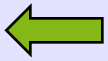
```
addi $29, $29, -36
sw $15, 0($29)
sw $16, 4($29)
sw $17, 8($29)
sw $18, 12($29)
sw $19, 16($29)
sw $20, 20($29)
sw $24, 24($29)
sw $25, 28($29)
sw $31, 32($29)
```

Rückschreiben der Registerinhalte

Epilog: Rückschreiben der Registerinhalte:

Prolog

Prozedurrumpf

Epilog 

Adr. von v	$4
n	$5
j	$17
Kopie von $4	$18
i	$19
Kopie von $5	$20
Hilfsvariable	$15,$16,$24,$25
Hilfsvariable, nicht gesichert	$8

```
lw   $15, 0($29)
lw   $16, 4($29)
lw   $17, 8($29)
lw   $18, 12($29)
lw   $19, 16($29)
lw   $20, 20($29)
lw   $24, 24($29)
lw   $25, 28($29)
lw   $31, 32($29)
addi $29, $29, 36
jr   $31
```

Zusammenfassung

- Nicht-verschachtelte Prozeduren
- jr- und jal-Befehle
- Das Stapel- (*stack*) Prinzip
- Verschachtelte Prozeduren
- Konventionen zur Registernutzung
- *Call-by-value*, *call-by-reference*
- Sichern und Rückschreiben von Registerinhalten
- *Callee-save* und *caller-save*
- Beispiele

2.2.2 Allgemeine Sicht der Befehlsschnittstelle

Bislang haben wir ein spezielles Beispiel einer Befehlsschnittstelle betrachtet.

☞ Jetzt allgemeine Sicht auf die Befehlsschnittstelle.

Zur Befehlsschnittstelle (***instruction set architecture, ISA***) gehören:

1. Die nutzbaren **Maschinenbefehle** (einschl. der elementaren Datentypen, siehe 2.2.2.2).

2. Ein **Modell des Speichers** (siehe 2.2.2.1).

3. Die logische Sicht auf das **Unterbrechungssystem** (*interrupt system,* siehe 2.2.2.3).

Vorlesung Rechnerstrukturen, Teil 2

2 Rechnerarchitektur

2.2 Die Befehlsschnittstelle

2.2.2 Allgemeine Sicht der Befehlsschnittstelle

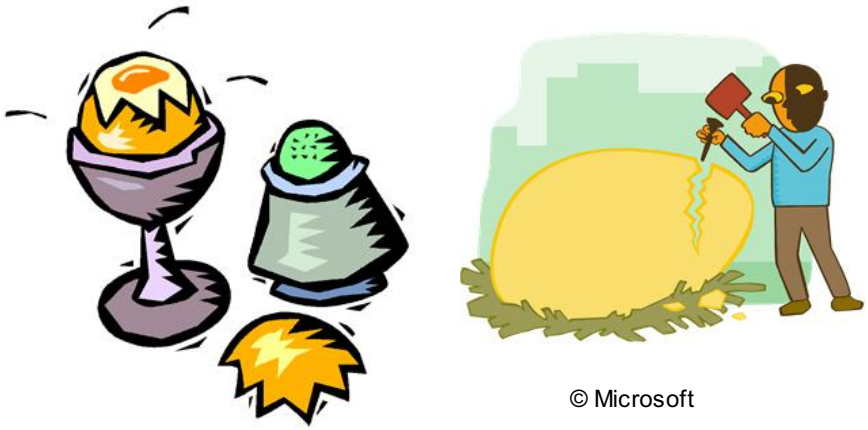
2.2.2.1 Das Speichermodell

Das Speichermodell

- **Zeitkomplexität des Zugriffs**: Verhalten entspricht dem eines großen Arrays, direkter Zugriff (ohne sequent. Suche)
- Größe der **adressierbaren Einheiten**: meist Byte-adressierbar.
- ***Alignment***-Beschränkungen beschreiben Einschränkungen hinsichtlich der Ausrichtung (z.B.: Befehle und *int's* beginnen auf Wortgrenzen).
- Maximale **Größe** des Speichers: Historisch gesehen immer wieder unterschätzt. "*640k will be enough for everyone",* Bill Gates (?), 1981. Verdopplung alle 2-3 Jahre.
- Die Speicher realisieren den **Zustand** des Rechensystems.

Little endians und *big endians* (1)
(*Endianess*)

Ursprung: Gulliver‘s Reisen: Ei am dicken oder dünnen Ende aufschlagen?



© Microsoft

Little endians: der am wenigsten signifikante Teil eines Wortes erhält die niedrigste Byteadresse.

Big endians: der signifikanteste Teil eines Wortes erhält die niedrigste Byteadresse.

Little endians und *big endians* (2)

Anwendungen, u.a.:
1. Bei der Speicherbelegung durch 32-Bit **`Integer`**
   Beispiel: Darstellung von 00010203_{16}

little endian		*big endian*	
Adresse	Wert	Adresse	Wert
"..00"	3	"..00"	0
"..01"	2	"..01"	1
"..10"	1	"..10"	2
"..11"	0	"..11"	3

Invertierung der beiden letzten Adressbits bewirkt Umschaltung zwischen den beiden Systemen.

Little endians und *big endians* (3)

Anwendungen, unter anderem:

2. Bei der Zeichenketten-Darstellung und Verarbeitung
   Beispiel: Darstellung von "Esel"

little endian		*big endian*	
Adresse	Wert	Adresse	Wert
"..00"	l	"..00"	E
"..01"	e	"..01"	s
"..10"	s	"..10"	e
"..11"	E	"..11"	l

Vorteile von *big endian*:

- Inkrementieren der Adresse liefert das nächste Zeichen.
- Verlängern der Zeichenkette ist einfach.

Little endians und *big endians* (4)

Anwendungen bei Datenübertragung: was wird zuerst übertragen?

- Internet: *Big-endian*:
  Signifikanteste Information wird zuerst übertragen
- RS-232 serielle Schnittstelle: *little-endian*:
  Am wenigsten signifikante Information wird zuerst übertragen („*LSB travels first*“)

Quellen (u.a.):

- Danny Cohen: *On Holy Wars and a Plea for Peace, IEEE Computer*, October 1981, (reine Text-Fassung: http://www.ietf.org/rfc/ien/ien137.txt)
- Bitte nicht den deutschen Wikipedia-Artikel!

Registerspeicher (1)

Warum Registerspeicher?

Man könnte in Befehlen 3 Adressteile angeben & bei arithmetischen Befehlen ausschließl. den Hauptspeicher benutzen.

Beispiel (so genannter 3-Adressbefehl):

```
add adresse1, adresse2, adresse3
```

Nachteile:

- Extrem große Befehlswortlänge
- Unterschiedlich lange Befehlsworte

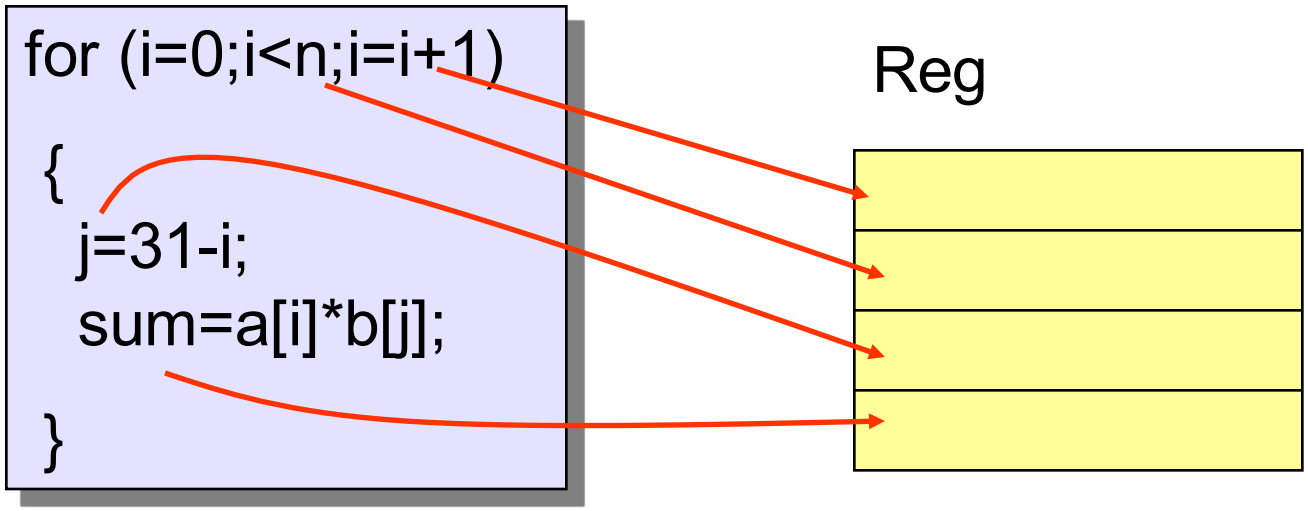
☞ Schnelle Abarbeitung in einem Fließband wird schwierig

Registeradressen ermöglichen Verkürzung der Adressteile.

☞ RISC: Alle Operanden & Ziele in Registern

Registerspeicher (2)

Zugriff auf Register viel schneller als Zugriff auf Hauptspeicher.
Programme verhalten sich in der Regel **lokal.**



lokaler Ausschnitt bietet schnellen Zugriff auf häufig genutzte Information.

Größe von Registerspeichern

Vorteile einer großen Anzahl von Registern:

Viele lokale Zugriffe möglich.

- schnelle Zugriffe.
- Entlastung des Hauptspeichers.

Nachteile einer großen Anzahl von Registern:

- Befehlswortbreite wächst an.
- Aufwand für das Retten und Rückschreiben (bei Prozeduraufrufen, Unterbrechungen, Prozessumschaltungen usw.) wächst an.

☞ Kompromiss

Homogene und heterogene Registersätze

Homogene Registersätze:
Alle Register besitzen dieselbe Funktionalität.

Heterogene Registersätze:
Manche Register besitzen spezielle Funktionen.
Beispiele: **`$0, $31, Hi`** und **`Lo,`**

Vorteile:

- Befehle mit vielen (≥ 4) Registern leicht zu realisieren.
- Weniger Registernummern im Befehl.
- Kann effizienter sein.

Nachteile:

- Separate Kopierbefehle erforderlich.
- Durch Compiler schwieriger zu nutzen.

Beispiele: 1. Motorola 680x0/Freescale ColdFire

Datenregister

D0
D1
D2
D3
D4
D5
D6
D7

Adressregister

A0
A1
A2
A3
A4
A5
A6

A7 *User stack pointer*
A7' *Interrupt stack pointer*
A7" *Master stack pointer*

PC
VBR *Vector base register*

Cache control register
Cache address register

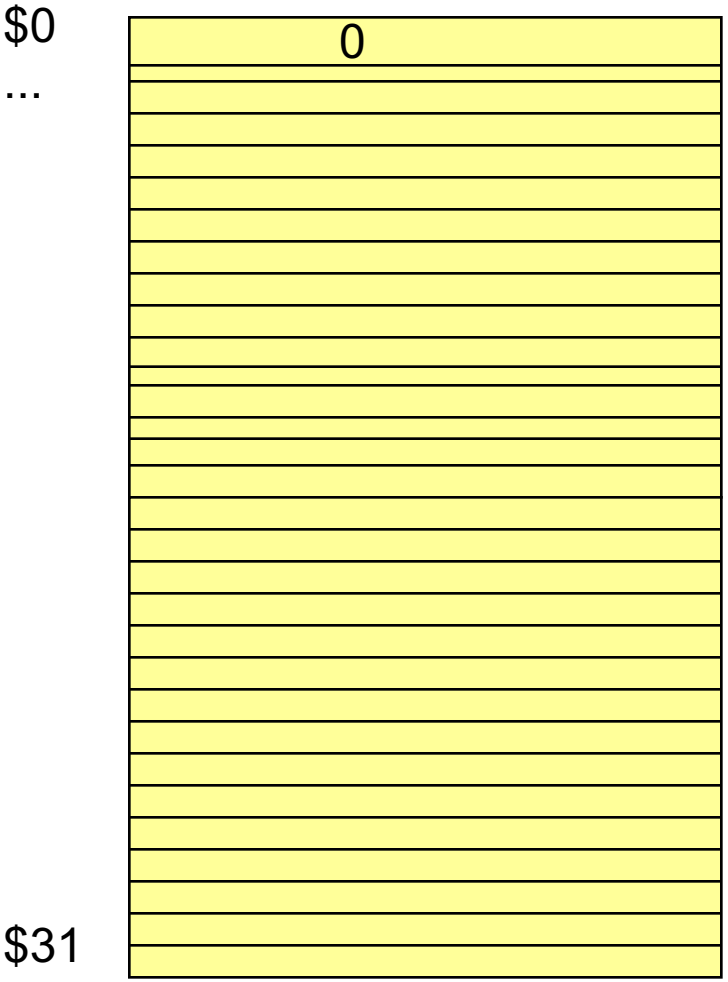
Trennung in Daten- & Adress-register bewirkt Reduktion auf 3 Bits/ Register-nummer; spezielle *stack pointer;* separates *condition code register (CCR)*

Statusregister

..
Extend
Negative
Zero
Overflow
Carry

Beispiele: 2. MIPS R2000-R8000

Registerspeicher



Hi
Lo
PC

Gleitkommaregister ...

So genannte *function register* (z.B. für Unterbrechungen)

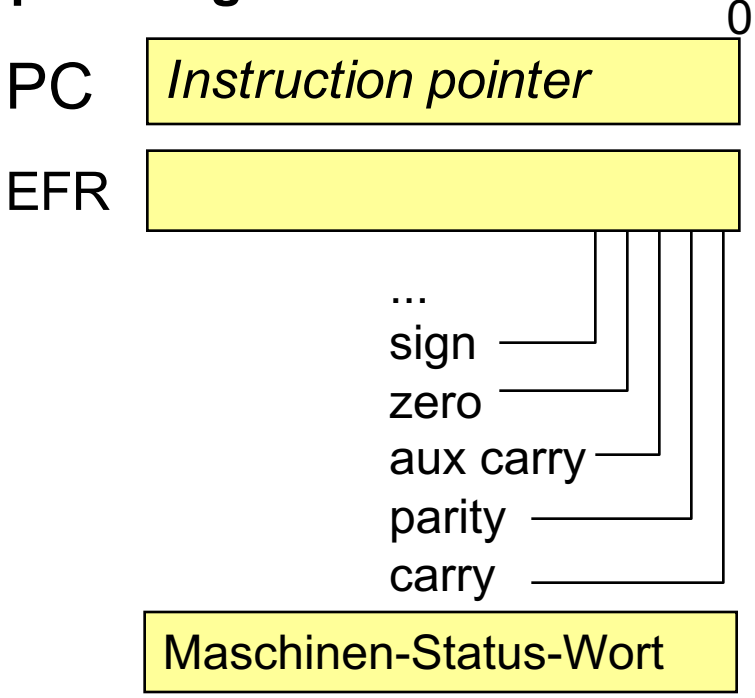
Keine *condition code register.*
Weitgehend homogen,
Ausnahmen: **`$0, $31, Hi, Lo`**,
function register,
Assemblerkonventionen für Registerbenutzung.

Beispiele: 3. Intel 80x86

Allgemeine Register

	0	
EAX	Arithm. Ergeb.	AX
EDX	& Ein/Ausgabe	DX
ECX	Zählregister	CX
EBX	Basis-Register	BX
EBP	Basis-Register	BP
ESI	Index-Register	SI
EDI	Index-Register	DI
ESP	Stack pointer	SP

Spezialregister



Segmentregister

CS	Code-Segment
SS	Stack-Segment
DS	Daten-Segment
ES	Daten-Segment
FS	Daten-Segment
GS	Daten-Segment

- Relativ kleiner & inhomogener Registersatz.
- Hohe Ansprüche an Geschwindigkeit des Hauptspeicherzugriffs.

☞ In neueren Prozessoren der Intel-Familie zusätzliche Register

Zusammenfassung

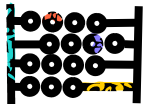
Allgemeine Sicht auf die Befehlsschnittstelle

- Speichermodell
  - Hauptspeicher
  - *Endianess*
  - Registerspeicher
- Befehlssatz
- Interrupts

2.2.2.2 Befehlssätze - Befehlsgruppen -

- Transferbefehle (**`lw, sw, mfhi`**)
- E/A-Befehle (**`in, out`**)

- Arithmetische Befehle (**`add, sub, mul, div`**)



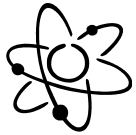
- Logische Befehle (**`and, or, not`**) 0101 v 0101
- Vergleichsbefehle (**`sgt`**, .. oder Seiteneffekt arithmet. Befehle)

- Bitfeld- und Flag-Befehle

- Schiebebefehle

- Sprungbefehle

- Kontrollbefehle (*disable interrupt*)

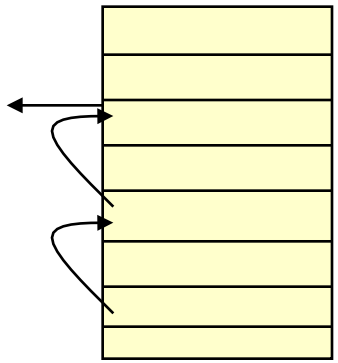


- Ununterbrechbare Befehle (*test-and-set*)

2.2.2.2 Befehlssätze
- Adressierungsarten -

Klassifikation der Adressierung nach der Anzahl der Zugriffe auf den Speicher

- **0-stufige Adressierung**
  z.B. Operanden in Registern, Direktoperanden
- **1-stufige Speicheradressierung (Referenzstufe 1)**
  z.B. bisherige **`lw`**, **`sw`**-Befehle
- **2-stufige Adressierung (Referenzstufe 2)**
- ***n*-stufige Adressierung (Referenzstufe *n*)**



0-stufige Adressierung

- **Registeradressierung:**
  ausschließlich Operanden aus & Ziele in Registern
  Beispiele:

`mflo $15`	`Reg[15]:=Lo`	MIPS
`clr,3`	`D[3]:=0`	680x0
`ldpsw,3`	`D[3]:=PSW`	680x0

- **unmittelbare Adressierung, Direktoperanden, *immediate addressing:***
  Operanden sind Teil des Befehlsworts
  Beispiele:
  - `lui $15,3` `Reg[15]:=3<<16` MIPS
  - `ld D3,#100` `D[3]:=100` 680x0

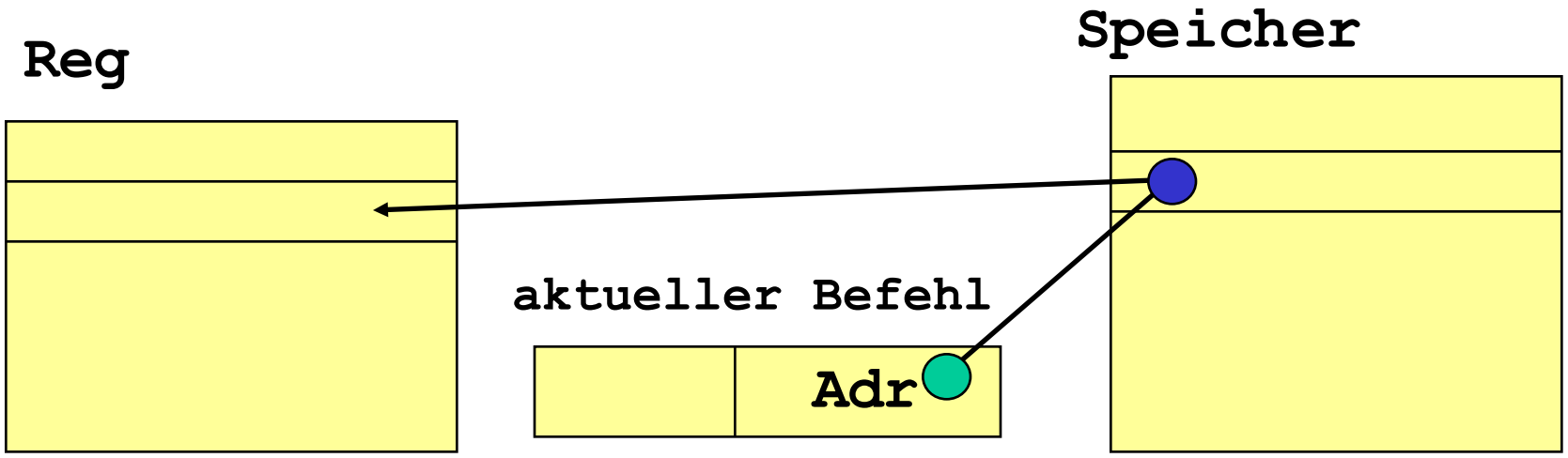
1-stufige Adressierung, Referenzstufe 1 (1)

Genau 1 Zugriff auf den Speicher

Beispiel:

```
lw $rega,Adr($regb) # Reg[$rega]:=Speicher[Adr+$regb]
```

1. **Fall $regb=0: Direkte oder absolute Adressierung**
   Adresse ist ausschließlich im Befehlswort enthalten



1-stufige Adressierung, Referenzstufe 1 (2)

2. Fall Adr=0: Register-indirekte Adressierung
Adresse ist ausschließlich im Register enthalten

Beispiele:

```
lw $15,($2)    # Reg[15]:=Speicher[Reg[2]]    MIPS
ld D3,(A4)     # D[3]:=Speicher[A[4]]          680x0
```

Varianten: *pre/post- increment/decrement* zur Realisierung von Stapeloperationen.
Beispiel:

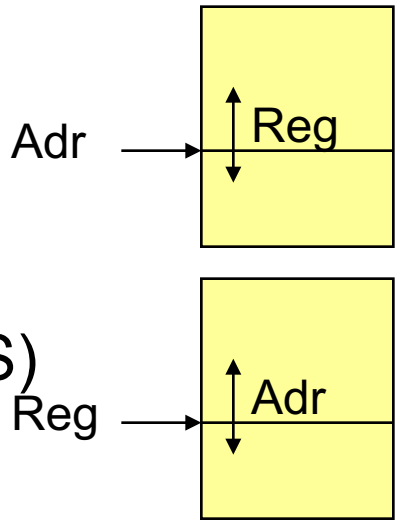
```
ld D3,(A4)+  # D[3]:=Speicher[A[4]];
             # A[4]:=A[4]+4 beim Laden von 32 Bit
```

1-stufige Adressierung, Referenzstufe 1 (3)

3. Fall Adr≠0, $regb≠0:
Relative, indizierte oder Basis-Adressierung

Varianten (nicht immer einheitlich bezeichnet):

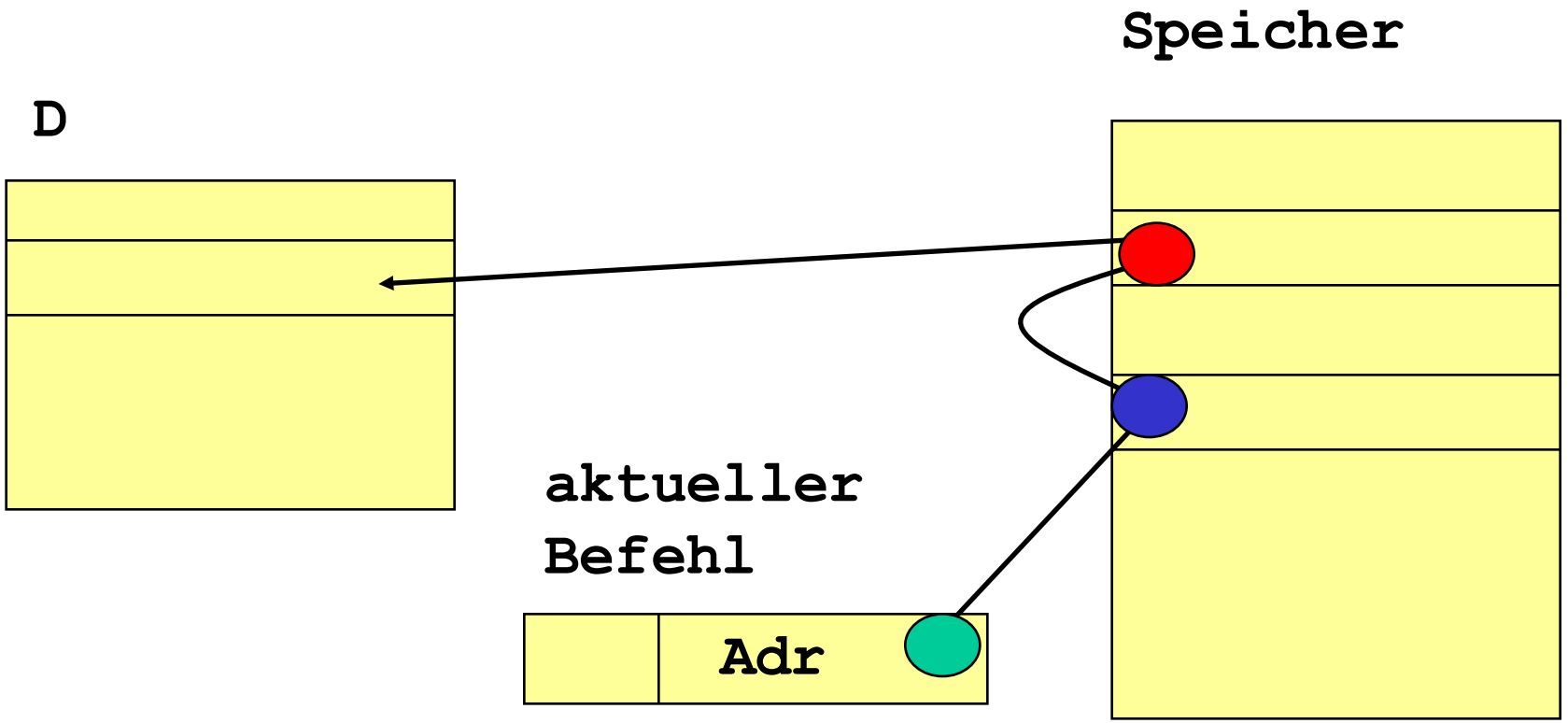
- **Indizierte Adressierung:**
  `Adr` umfasst vollen Adressbereich,
  Register evtl. nicht.
- **Basisadressierung,**
  Register-Relative Adressierung (MIPS)
  Register umfasst vollen Adressbereich,
  `Adr` evtl. nicht.



2-stufige Adressierung - Indirekte Adressierung - (1)

- **Indirekte (absolute) Adressierung**

`D[Reg]:=Speicher[Speicher[Adr]]`



2-stufige Adressierung - Indirekte Adressierung - (2)

- **Indirekte Register-indirekte Adressierung**

  `D[Reg]:=Speicher[Speicher[D[IReg]]` mit `IReg∈0..7`

- **Vorindizierung**

  `D[Reg]:=Speicher[Speicher[Adr+D[IReg]]` mit `IReg∈0..7`

- **Nachindizierung**

  `D[Reg]:=Speicher[Speicher[Adr]+D[IReg]]` mit `IReg∈0..7`

- **Indirekte Programmzähler-relative Adressierung**

  `D[Reg]:=Speicher[Speicher[Adr+PC]]`

  `PC:=Speicher[Adr+PC+i],` mit `i∈{1,2,4}`

n-stufige Adressierung

- Die fortgesetzte Interpretation des gelesenen Speicherwortes als Adresse des nächsten Speicherworts nennt man **Dereferenzieren.**
- Referenzstufen $n > 2$ werden nur in Ausnahmefällen realisiert.
- Fortgesetztes Dereferenzieren ist u.a. zur Realisierung der logischen Programmiersprache PROLOG mittels der *Warren Abstract Machine* (WAM) wichtig.
- Bei der WAM wird die Anzahl der Referenzstufen durch Kennzeichen-Bits in den gelesenen Speicherworten bestimmt.

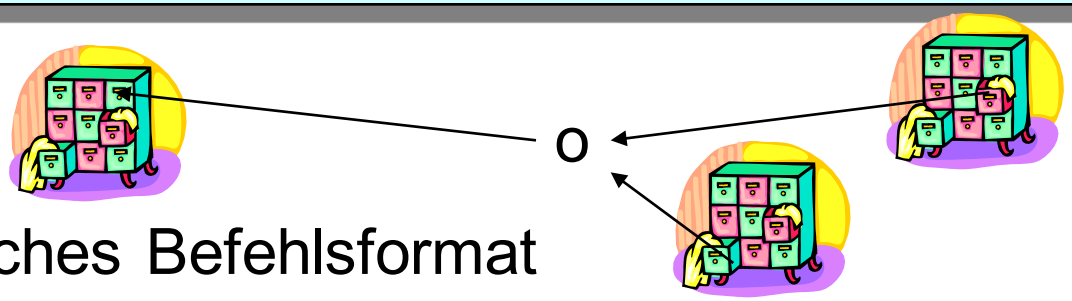
Übersicht

- 0-stufige Adressierung
  - Register-Adressierung
  - unmittelbare Adressierung
- 1-stufige Speicheradressierung (Referenzstufe 1)
  - direkte Adressierung, absolute Adressierung
  - Register-indirekte Adressierung
  - indizierte Adressierung, Basisadressierung
  - Programmzähler-relative Adressierung
- 2-stufige Adressierung (Referenzstufe 2, indirekte A.)
  - absolute Adressierung
  - Register-indirekte Adressierung
  - indizierte Adressierung
  - Programmzähler-relative Adressierung
- n-stufige Adressierung (Referenzstufe n>2)

n-Adressmaschinen
- Die 3-Adressmaschine (1) -

Klassifikation von Befehlssätzen bzw. Befehlen nach der Anzahl der Adressen bei 2-stelligen Arithmetik-Befehlen.

- **3-Adressmaschinen, 3-Adressbefehle**
  IBM-Bezeichnung „SSS“; Befehle bewirken Transfer:
  `Speicher[s1] := Speicher[s2] o Speicher[s3]`
  `s1,s2,s3:` Speicheradressen, **`o:`** 2-stellige Operation.



Mögliches Befehlsformat

Opcode	s1	s2	s3

4x 32=128 Bit Befehlswortbreite

- Die 3-Adressmaschine (2) -

Anwendung: Zerlegung der Anweisung

D= (A+B)*C;

Mit a=Adresse von A, b=Adresse von B, usw.:

```
add t1, a,b   #Speicher[t1]:=Speicher[a]+Speicher[b]
mult d,t1,c   #Speicher[d]:=Speicher[t1]*Speicher[c]
```

Programmgröße: 2*128=256 Bit

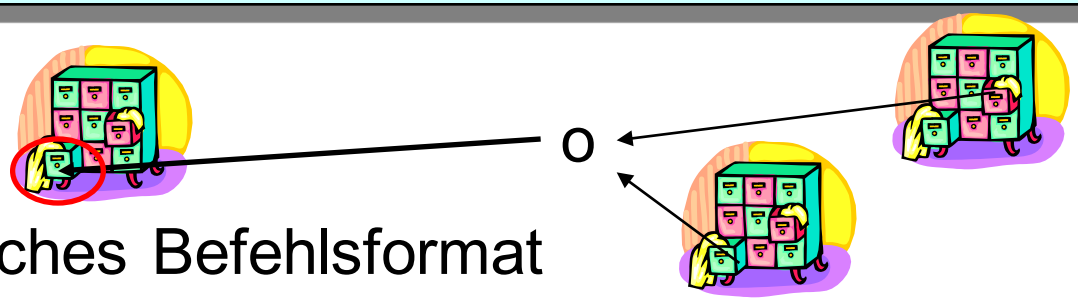
Speicherzugriffe: 2*3=6

- 3-Adressbefehle im Compilerbau als Zwischenschritt benutzt.
- MIPS: 3-Adress-RRR-Format für Arithmetikbefehle.

Die 2-Adressmaschine (1)

Verkürzung des Befehlswortes durch Überschreiben eines Operanden mit dem Ergebnis.

- **2-Adressmaschinen, 2-Adressbefehle**
  IBM-Bezeichnung „SS“; Befehle bewirken Transfer der Form:
  `Speicher[s1] := Speicher[s1] o Speicher[s2]`
  mit **`s1,s2:`** Speicheradressen, **`o:`** 2-stellige Operation.



Mögliches Befehlsformat

Opcode	s1	s2

3x 32=96 Bit Befehlswortbreite

Die 2-Adressmaschine (2)

Anwendung: Zerlegung der Anweisung

D= (A+B)*C;

Mit a=Adresse von A, b=Adresse von B, usw.:

```
move  t1,a     #Speicher[t1]:=Speicher[a]
add   t1,b     #Speicher[t1]:=Speicher[t1]+Speicher[b]
mult  t1,c     #Speicher[t1]:=Speicher[t1]*Speicher[c]
move  d,t1     #Speicher[d]:=Speicher[t1]
```

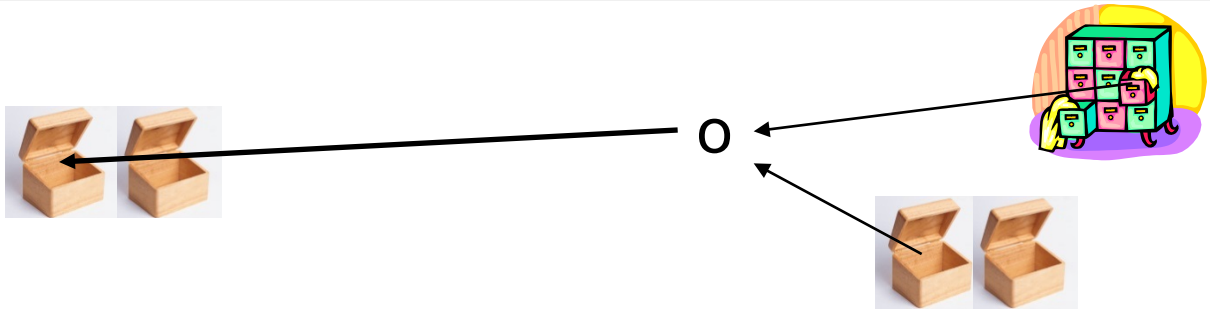
Programmgröße: 4*96=384 Bit

Speicherzugriffe: 2*3+2*2=10

Die 1½-Adressmaschine (1)

Weitere Verkürzung des Befehlswortes mit Registerspeichern

- **1½-Adressmaschinen, 1½-Adressbefehle**
  IBM-Bezeichnung „RS“; Befehle bewirken Transfer der Form:
  `Reg[r]:= Reg[r] o Speicher[s]`
  `s:` Speicheradresse, **`r`**: Registernummer, **`o:`** Operation



Mögliches Befehlsformat

Opcode	RegNr.	s

2x 32=64 Bit Befehlswortbreite

Enthält zusätzlich RR-Befehle der Wirkung Reg[r1]:= Reg[r2]

Die 1½-Adressmaschine (2)

Anwendung: Zerlegung der Anweisung

D= (A+B)*C;

Mit a=Adresse von A, b=Adresse von B, usw.:

```
lw,   $8, a    # Reg[8]:=Speicher[a]
add   $8, b    # Reg[8]:=Reg[8] + Speicher[b]
mult $8, c    # Reg[8]:=Reg[8] * Speicher[c]
sw    $8, d    # Speicher[d]:=Reg[8]
```

Programmgröße: 4*64=256 Bit

Speicherzugriffe: 4

Die 1-Adressmaschine (1)

Sonderfall der Nutzung von nur 1 Register („Akkumulator")

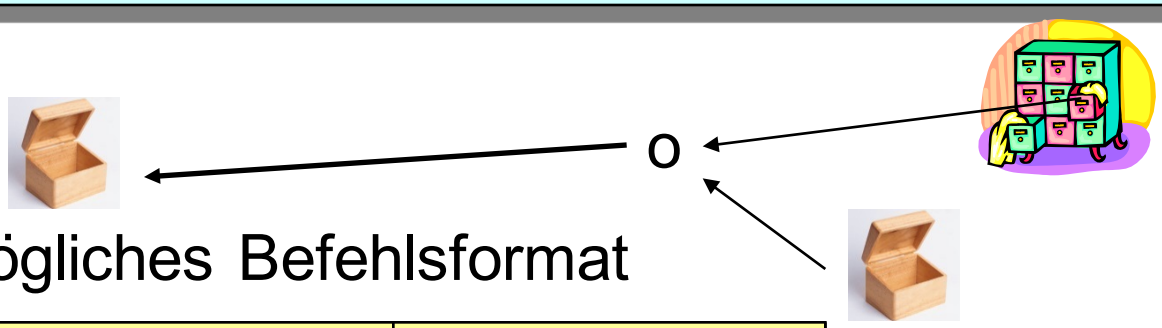
- **1-Adressmaschinen, 1-Adressbefehle**

Befehle bewirken Transfer der Form:

`accu:= accu o Speicher[s]`

mit **`s:`** Speicheradresse

`o:` 2-stellige Operation (wie +, -, *, /).



Mögliches Befehlsformat

Opcode .	s

2x 32=64 Bit Befehlswortbreite (i.d.R. kürzere Wortlänge)

Die 1-Adressmaschine (2)

Anwendung: Zerlegung der Anweisung

D= (A+B)*C;

Mit a=Adresse von A

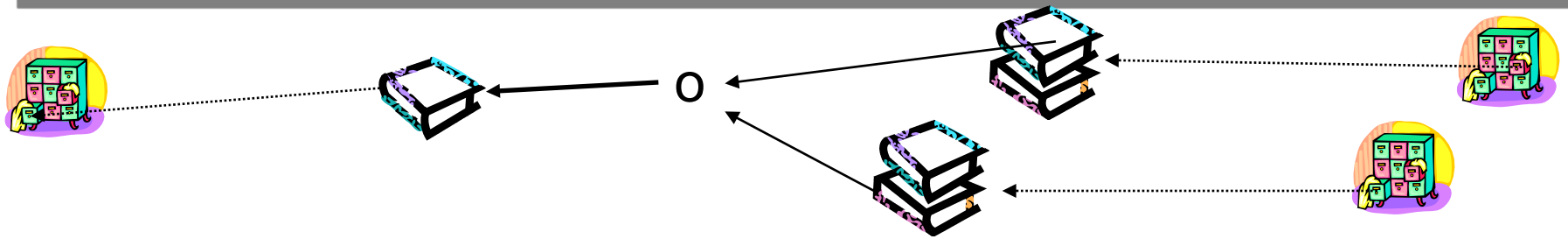
```
lw,   a    # accu :=Speicher[a]
add   b    # accu:=accu + Speicher[b]
mult  c    # accu:=accu * Speicher[c]
sw    d    # Speicher[d]:=accu
```

Programmgröße: 4*64=256 Bit

Speicherzugriffe: 4

Die 0-Adressmaschine (1)

Arithmetische Operationen werden auf Stapel ohne Angabe der Operanden ausgeführt. Die 2 obersten Stapelelemente werden verknüpft & durch das Ergebnis der Operation ersetzt.



- **0-Adressmaschinen, 0-Adressbefehle**

3 Sorten von Befehlen:

`push a` # **`Speicher[a]`** wird oberstes Stapelelement

`add` # Addiere die beiden obersten Stapelelemente und ersetze sie durch die Summe. Ähnlich für -,*,/

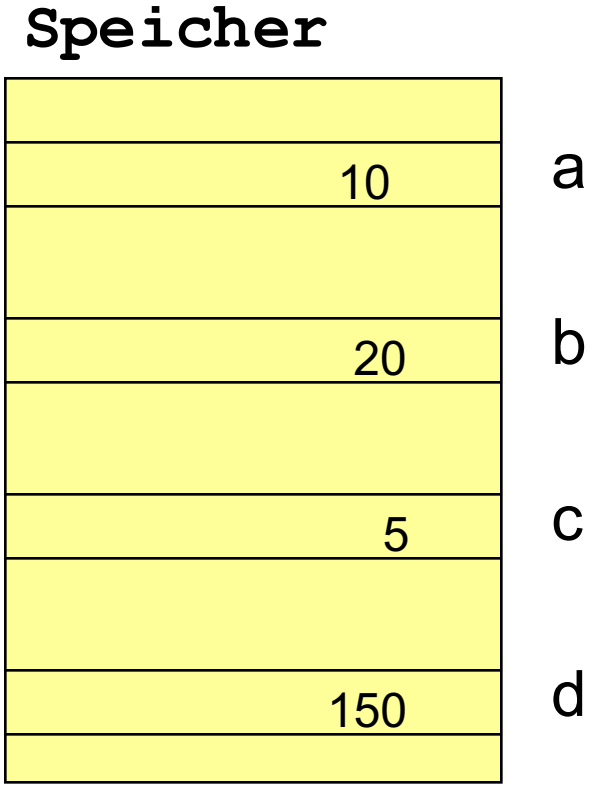
`pop a` # **`Speicher[a]:=`** oberstes Stapelelement. Entferne dieses Element vom Stapel.

Die 0-Adressmaschine (2)

Beispiel:

```
push a
push b
add
push c
mult
pop d
```

Stapel



Mögliche Befehlsformate:
Arithmetik-Befehle: 1 Byte;
push, pop: 1 Byte+Adresse=5 Byte

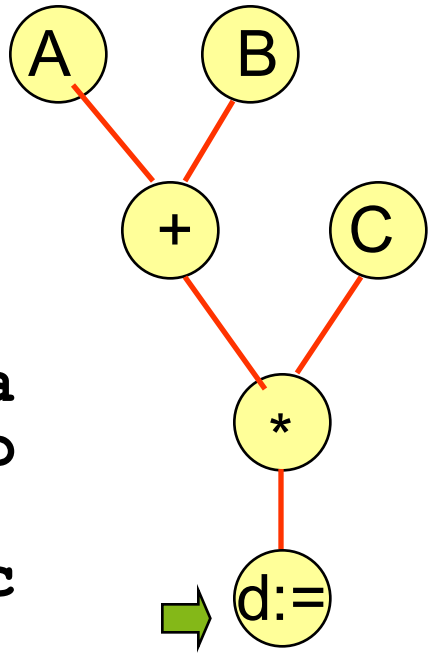
Programmgröße:
22 Byte=176 Bit,
4 Speicherzugriffe

Die 0-Adressmaschine (3)

Wie kommt man vom Ausdruck D= (A+B)*C;
zur Befehlssequenz?

1. Verwendung der **postfix**-Notation:
   AB+C*

2. Mittels des Ausdrucksbaums:
   GenerateCode(p)
   {
   gehe zum linken Sohn;
   gehe zum rechten Sohn;
   erzeuge Befehl für aktuellen Knoten;
   }




```
push a
push b
add
push c
mult
pop d
```

Virtuelle Java-Maschine ist eine 0-Adressmaschine

Zusammenfassung

Allgemeine Sicht auf die Befehlsschnittstelle

- Speichermodell
  - Hauptspeicher
  - *Endianess*
  - Registerspeicher
- Befehlssatz (*instruction set architecture, ISA*)
  - Befehlsgruppen
  - Referenzstufen
  - n-Adressmaschinen
  - RISC vs. CISC-Maschinen
- Interrupts

Klassifikation von Befehlssätzen
- *Reduced instruction set computers (RISC)* (1)-

Wenige, einfache Befehle wegen folgender Ziele:

- Hohe Ausführungsgeschwindigkeit
  - durch Fließbandverarbeitung (siehe 2.3.3)
  - durch kleine Anzahl interner Zyklen pro Befehl

Def.: Unter dem **CPI-Wert** (engl. ***cycles per instruction***) einer Menge von Maschinenbefehlen versteht man die mittlere Anzahl interner Bus-Zyklen pro Maschinenbefehl.

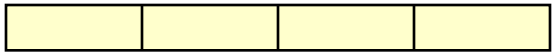
RISC-Maschinen: CPI möglichst nicht über 1.
CISC-Maschinen (s.u.): Schwierig, unter CPI = 2 zu kommen.

Programmlaufzeit = Anzahl der auszuführenden Befehle * CPI-Wert des Programms * Dauer eines Buszyklus

Klassifikation von Befehlssätzen
- *Reduced instruction set computers (RISC)* (2)-

Eigenschaften daher:

- feste Befehlswortlänge



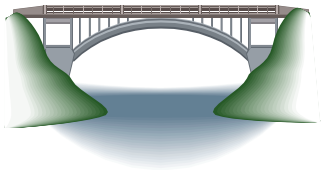
- LOAD/STORE-Architektur

ld $2,..; ld $3,..; add $3,$3,$2; sw $3

- einfache Adressierungsarten

z.B. keine indirekte Adr.

- „semantische Lücke“ zwischen Hochsprachen & Assemblerbefehlen durch Compiler überbrückt.



- Statt aufwändiger Hardware zur Beseitigung von Besonderheiten (z.B. 256 MB-Grenze bei MIPS, 16 Bit Konstanten) ☞ Aufgabe der Software
- Rein in Hardware realisierbar („mit Gattern und Flip-Flops“)

Complex instruction set computers (CISC)

- ***Complex instruction set computers (CISC)***

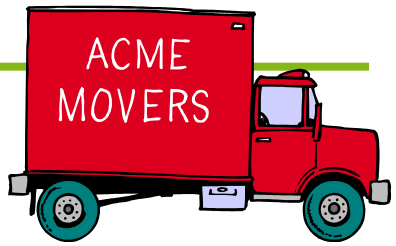
Entstanden in Zeiten schlechter Compiler & großer Geschwindigkeitsunterschiede Speicher / Prozessor

☞ Befehle sollten möglichst nahe an den Hochsprachen sein (keine semantische Lücke)

☞ Mit jedem geholten Befehl sollte der Prozessor viel tun

☞ sehr komplexe Befehle

Complex instruction set computers (CISC)
Beispiel MC680x0 (1)



Beispiel: Motorola 68000 (erster Prozessor der 680x0-Serie)

Format des Kopierbefehls MOVE:

Opcode	Größe	Ziel		Quelle	
"00"	"01"=Byte, "11"=Wort, "10"=Doppelwort (32 Bit)	Register	Modus	Register	Modus
bis zu 4 Erweiterungsworte zu je 16 Bit					

Viele komplexe Adressierungsarten schon in den ersten Prozessoren der Serie.

Complex instruction set computers (CISC)
- Beispiel MC680x0 [☞ ColdFire] (2) -

Modus	Register feld	Erweit.	Notation	Adressierung
"000"	n	0	Dn	Register-Adressierung
"001"	n	0	An	Adressregister-Adressierung
"010"	n	0	(An)	Adressregister indir.
"011"	n	0	(An)+	Adressreg. indirekt.mit *postincrement*
"100"	n	0	-(An)	Adressreg. indirekt.mit *predecrement*
"101"	n	1	d(An)	Relative Adressierung mit 16 Bit Distanz
"110"	n	1	d(An,Xm)	Register-relative Adressierung mit Index
"111"	"000"	1	d	direkte Adressierung (16 Bit)
"111"	"001"	2	d	direkte Adressierung (32 Bit)
"111"	"010"	1	d(*)	Programmzähler-relativ
"111"	"011"	1	d(*,Xn)	Programmzähler-relativ mit Index
"111"	"100"	1-2	#zahl	unmittelbare Adressierung

Complex instruction set computers (CISC)
- Eigenschaften -

- Relativ kompakte Codierung von Programmen
- Für jeden Befehl wurden mehrere interne Zyklen benötigt
  - ☞ Die Anzahl der Zyklen pro Befehl (der ***cpi***-Wert) war groß
- (Mikro-) Programm zur Interpretation der Befehle nötig
- Compiler konnten viele Befehle gar nicht nutzen

Zusammenfassung

Allgemeine Sicht auf die Befehlsschnittstelle

- Speichermodell
  - Hauptspeicher
  - *Endianess*
  - Registerspeicher
- Befehlssatz (*instruction set architecture, ISA*)
  - Befehlsgruppen
  - Referenzstufen
  - n-Adressmaschinen
  - RISC vs. CISC-Maschinen
- Interrupts

2 Teil 2 des Kurses Rechnerstrukturen

2.2 Die Befehlsschnittstelle

2.2.2 Allgemeine Sicht der Befehlsschnittstelle

2.2.2.3 Unterbrechungen

Unterbrechungen

„Während der normalen Bearbeitung eines Programms kann es .. zu **Ausnahme-Situationen** (engl. *exceptions*) kommen, die vom Prozessor aus eine vorübergehende Unterbrechung oder aber einen endgültigen Abbruch des Programms, … verlangen.“ (Zitat Bähring)

Beispiele:

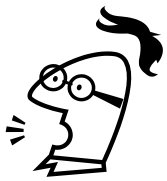
- arithmetische Ausnahmen (div 0),
- Überläufe (sofern Ausnahmebehandlung eingeschaltet)
- ungültige Adresse
- Speicherschutz verletzt
- ...

Ablauf (bei MIPS)

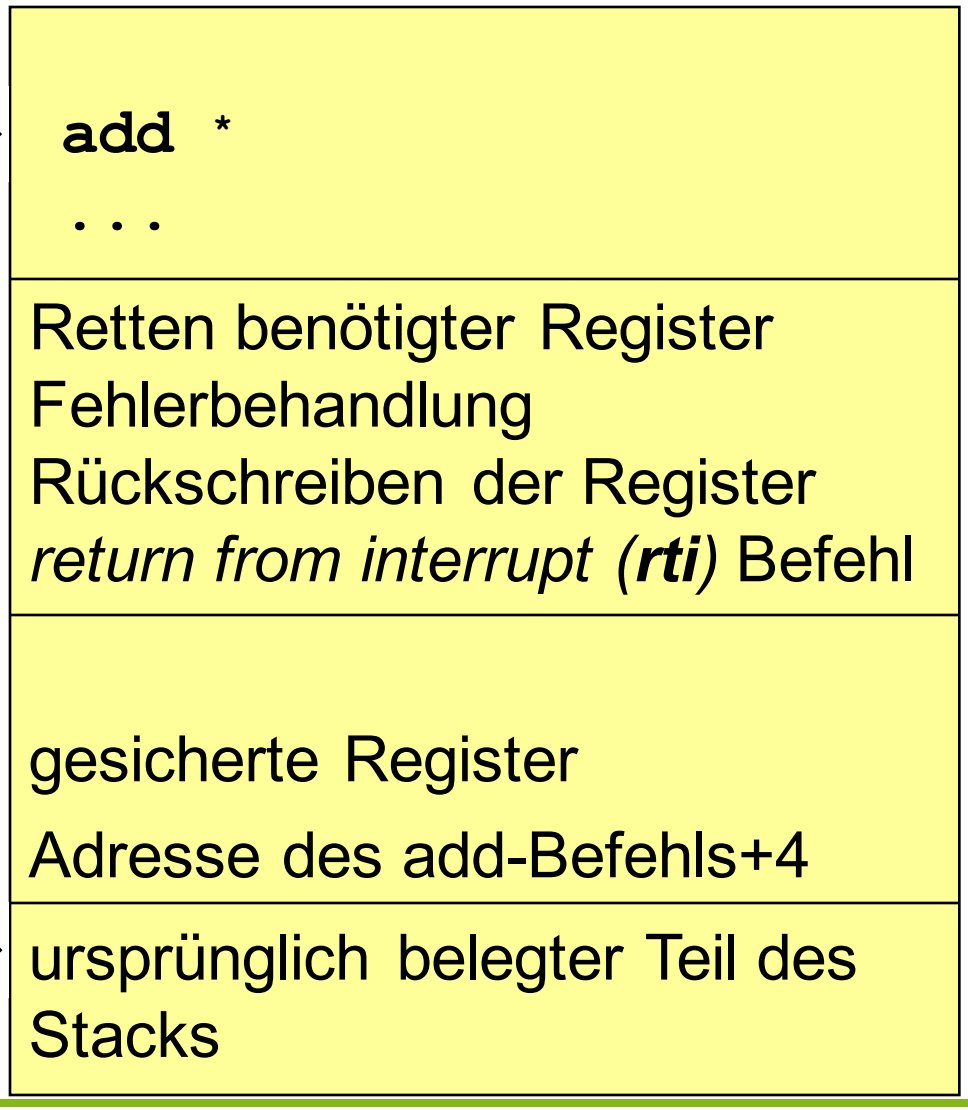
1. Benutzer könnte *Stack-Pointer* ungültig setzen
2. Überschreiben des *Stack* kann Überraschung sein.

☞ SPIM-traphandler sichert in festen Speicherzellen!

Besser wäre ein separater *Interrupt-Stack* (wie MC68k).



PC →



$29 →

* bei div prüfen MARS + SPIM per SW auf div 0

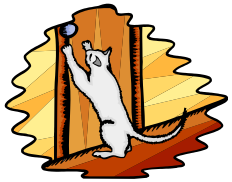
Interrupts und Prozeduren (1)

- Interrupts entsprechen automatischem Aufruf von Prozeduren bei Fehlern oder Ausnahmen.
- Hardwaremäßig erzeugter Aufruf entspricht einem **`jal`**-Befehl;
  - Zieladresse bestimmt sich aus der Unterbrechungsursache
    - ***vectored interrupt :***
      Ursache mit der Unterbrechung direkt bekannt
    - **sonst:** Ursache mit Statusabfragen bestimmen!
  - Rückkehradresse wird sofort auf dem Stapel° gespeichert.
- Benutzung des Stapels erlaubt verschachtelte, sogar rekursive* Unterbrechungen

° Alte Rechner benutzen eine feste Zelle pro Ursache (☞ keine Rekursion).
* Wegen Problemen in der Realisierung meist ausgeschlossen.

Interrupts und Prozeduren (2)

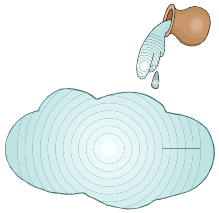
- Bei Vorliegen mehrerer Unterbrechungsursachen entscheidet deren **Priorität**, welche Ausnahmebehandlung zuerst gestartet wird.
- Interrupts können meist mit besonderen Befehlen **gesperrt** werden.



- Dem `jr $31` entspricht hier der `rti`-Befehl,
- Rückkehr-Adresse wird dem Stack entnommen

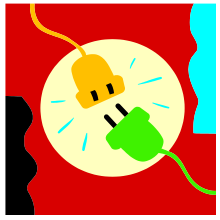
Synchrone und asynchrone Unterbrechungen

- Ausnahmesituationen können durch das aktuell ausgeführte Programm herbeigeführt werden.
  ☞ **synchrone Unterbrechungen**.
  - Beispiel: Arithmetik-Überlauf



- Ausnahmesituationen können ohne Bezug zum aktuellen Programm sein. Ursachen: externe Systemkomponenten wollen vom Prozessor „bedient“ werden.
  ☞ **asynchrone Unterbrechungen**.

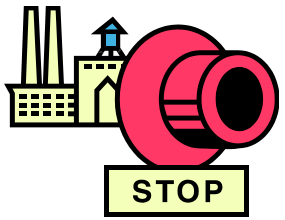
  Beispiel: Netzausfall



Unterbrechung am Ende des laufenden Befehls oder bereits während der Bearbeitung ?

- In einigen Fällen ist ein Abbruch des laufenden Befehls erforderlich:

  Beispiele: Speicherschutzverletzung, illegaler Befehl

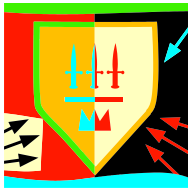


- In anderen Fällen kann bis zum Ende der Bearbeitung des laufenden Befehls gewartet werden:

  Beispiele: alle asynchronen Unterbrechungen (außer bei sehr langer Bearbeitungszeit des Befehls, wie z.B. beim Kopieren von großen Blöcken mit einem Befehl).

Endgültiger Abbruch oder Fortfahren ?

- In einigen Fällen ist ein endgültiger Abbruch des laufenden Befehls erforderlich: Beispiele:
  - Speicherschutzverletzung,
  - illegaler Befehl

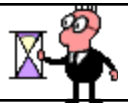
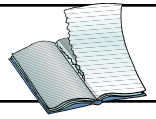


- In anderen Fällen ist ein Fortfahren in der Befehlsausführung erforderlich: Beispiel:
  - Automatische Seiteneinlagerung von der Platte

Übersicht über verschiedene Unterbrechungen

Klasse	Synchron?	Unterbrechung im Befehl ?
Ein-/Ausgabegerät	nein	nein
Betriebssystems-Aufruf	ja	nein
TRACE-Befehl	ja	nein
Überlauf	ja	ja
Timer	nein	nein
Seitenfehler	ja	ja (!)
Schutz-Verletzung	ja	ja (!)
Unimplementierter Befehl	ja	ja
Hardware-Fehler	beides	ja
Netzausfall	nein	nein (?)





Systemaufrufe in der MIPS-Maschine

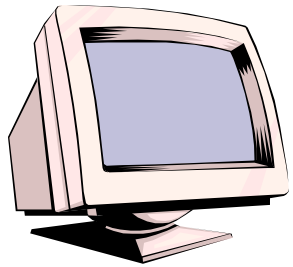
Aufruf von Funktionen des Betriebssystems per ***supervisor call*** (SVC),

- Parameter in Registern

MIPS/MARS: syscall, Servicenr. in $v0 (=$2)			
Servicename	**Servicenr**	**Argumente**	**Ergebnis**
print_int	1	$a0 = integer	-
print_string	4	$a0 = Adresse der Zeichenkette	-
read_int	5	-	$v0
read_string	8	$a0: Pufferadresse, $a1: Pufferlänge	-
exit	10		-

Beispiel der Anwendung

```
        .data
str:    .asciiz "die Antwort ist: " # Nullbyte am Ende
        .text
        li $v0, 4           # Nummer des Systemaufrufs
        la $a0, str         # Adresse der Zeichenkette
        syscall             # Systemaufruf
        li $v0, 1           # Nummer des Systemaufrufs
        li $a0, 42          # integer
        syscall             # Systemaufruf
        li $v0, 10          # exit
        syscall
```



- E/A bezieht sich hier auf das Terminal.
- MARS realisiert weitere SVC-Aufrufe, u.a. *file-I/O* (s. *Online*-Doku)

Realisierung von Systemaufrufen

- „Normalerweise”: Aufruf eines hardwaremäßig nicht implementierten Befehls (MIPS: 0x0000000c);
  Hardware startet Unterbrechungsbehandlung;
  In der Behandlung: Erkennung des SVC-Aufrufs;
  Verzeigung an Routine innerhalb des *kernels*.

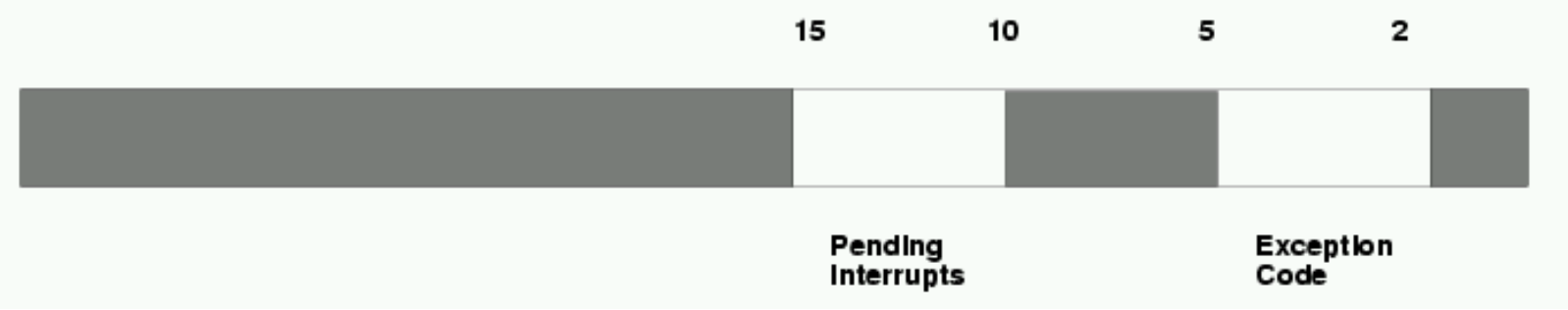
- Beim Simulator: Simulation im Simulator selbst; dadurch keine Verzweigung in den *kernel*-Bereich erkennbar.

Unterbrechungen bei MIPS/MARS

- Im Falle einer Ausnahme erfolgt ein Sprung an die Adresse 0x80000180
- Ein an dieser Adresse gespeicherter *Trap-Handler* muss die Unterbrechungsursache erfragen. Hierzu müssen Register des Coprozessors 0 gelesen werden.
- Hierzu vorhandene Befehle:
  mfc0, mtc0, lwc0, swc0.

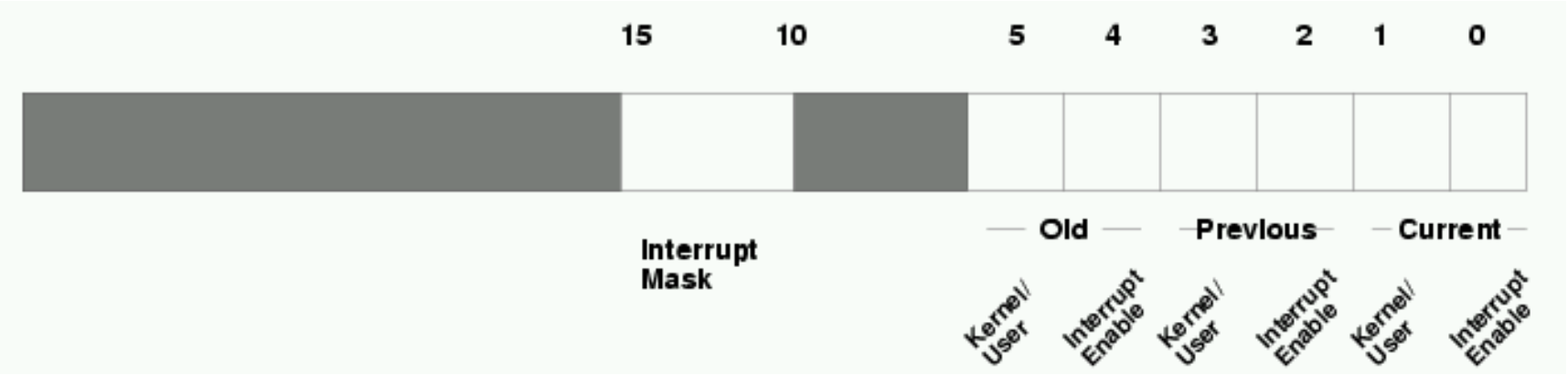
Register Name	Nummer	Benutzung
BadVAddr	8	*Memory address of address exception*
Status	12	*Interrupt mask and enable bits*
Cause	13	*Exception type & pending interrupt bits*
EPC	14	*Address of instr. that caused exception*

Coprozessorregister-Bedeutung
- Cause-Register-



Number	Name	Description
0	INT	External interrupt
4	ADDRL	Address error exception (load or instruction fetch)
5	ADDRS	Address error exception (store)
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data load or store
8	SYSCALL	Syscall exception
9	BKPT	Breakpoint exception
10	RI	Reserved instruction exception
12	OVF	Arithmetic overflow exception

Coprozessorregister-Bedeutung
- Status-Register-

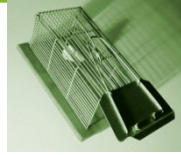


Wiedereintrittsfähiger Code *(reentrant code)*

- Während Unterbrechungsbehandlung weitere Unterbrechung möglich (Beispiel: Schutzverletzung während einer Unterbrechungsbehandlung).
- Entspricht verschachtelten Prozeduraufrufen; Registerinhalte und Variablen auf Stapel speichern!
- Unterbrechungsbehandlungen, die selbst wieder unterbrochen werden können, heißen **wiedereintrittsfähig** (engl. *re-entrant* bzw. *reentrant*).

- Standard-*Trap*-Handler des SPIM ist nicht *reentrant*, (Register werden in festen Speicherzellen gesichert).

SPIM-*Traphandler*



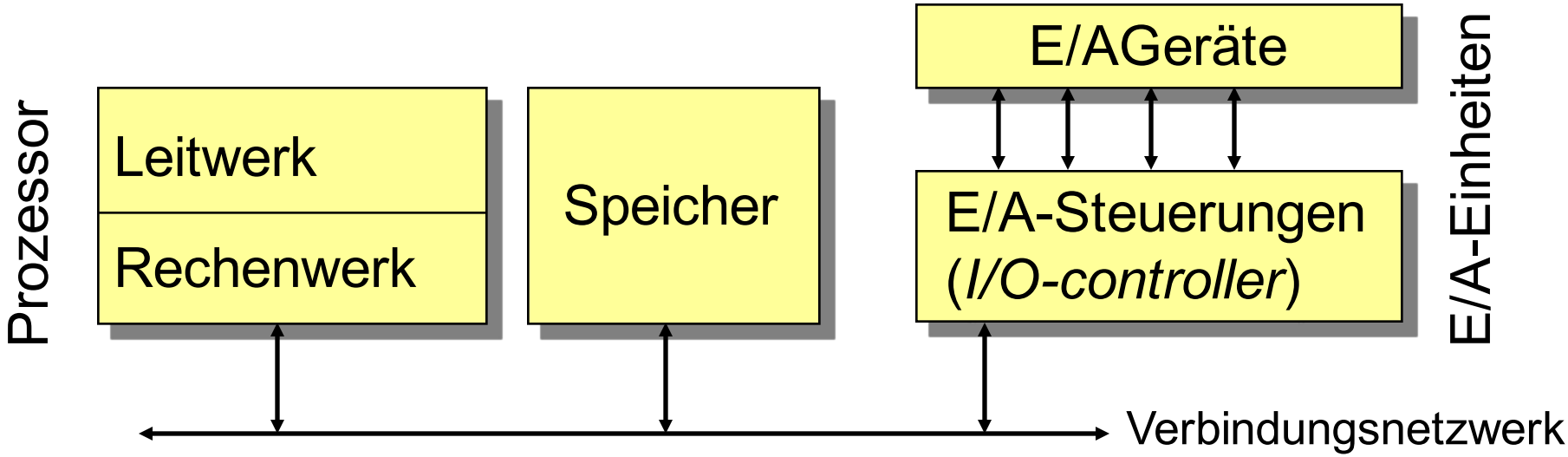
neu

```
s1:  .word 0            # Speicher zum Sichern von Registern;
s2:  .word 0            # dto.
 .ktext 0x80000180      # Code gehört zum Kernel Textsegment
 .set noat              # keine Nutzung von $1 durch Assembler
 move $k1 $at           # Retten von $1 in $27 (für Kernel)
 .set at                # Nutzung durch von $1 wieder möglich
 sw $v0 s1              # Rette $v0 in feste Speicherzelle
 sw $a0 s2              # Rette $a0 in feste Speicherzelle
                        # Nutzung von $1=$at,$v0,$a0,$k0 möglich
 mfc0 $k0 $13           # Lade Unterbrechungsursache
 ...                    # Verzweige abhängig von $k0
 ...                    # Lösche Ursachenregister
 lw $v0 s1              # Rückschreiben von $v0
 lw $a0 s2              # Rückschreiben von $a0
 .set noat
 move $at $k1           # Restore $at
 .set at
 rfe                    # Restaurieren von Statusregistern
 mfc0 $k0 $14           # Hole alten Wert des PC
 addiu $k0 $k0 4        # Erhöhe um 4
 jr $k0                 # Führe unterbrochenes Programm fort
```

Das Von Neumann-Modell

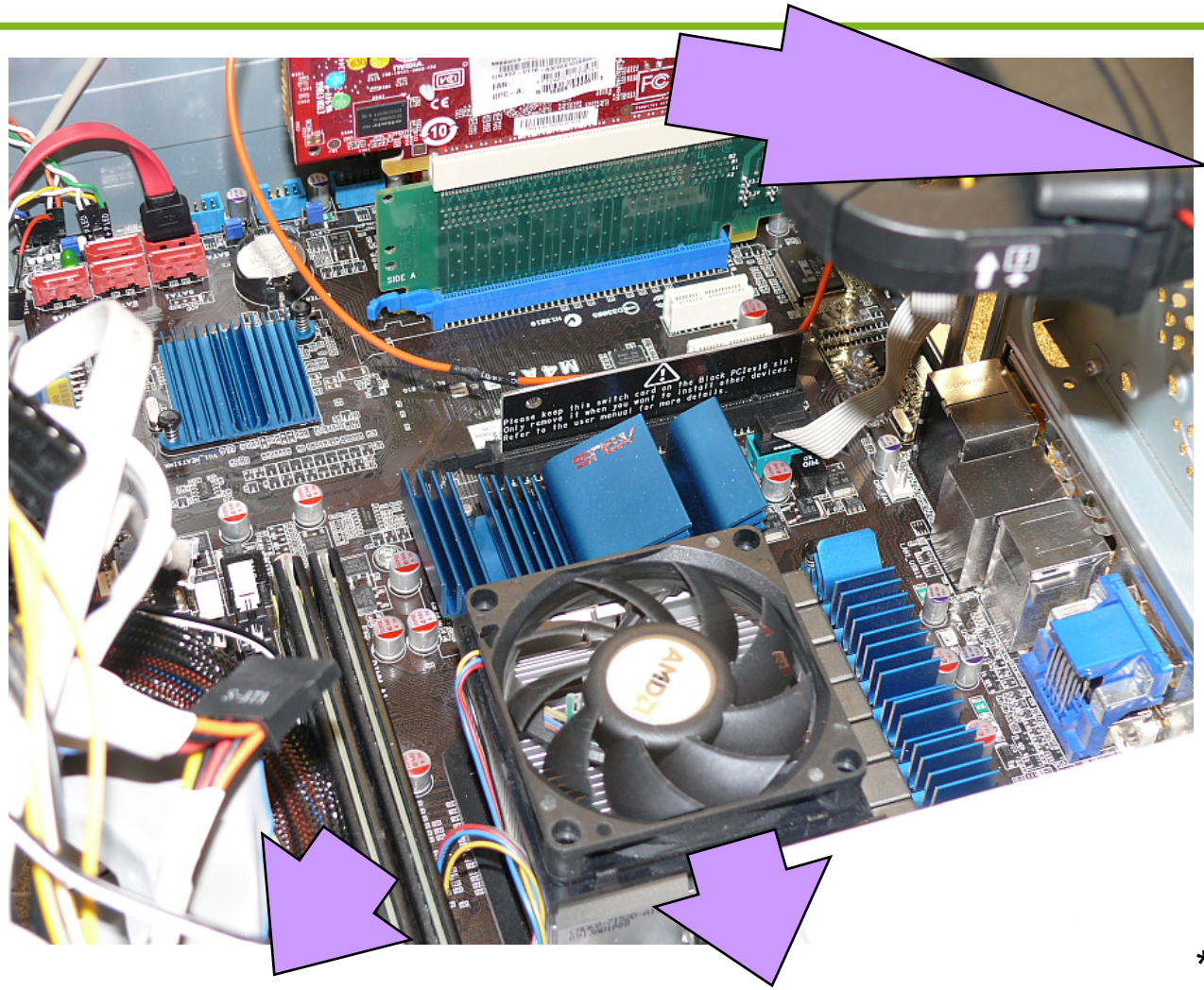
Fast alle der heute üblichen Rechner gehen auf den **Von Neumann-Rechner** mit folgenden Eigenschaften zurück:

1. Die Rechenanlage besteht aus den Funktionseinheiten **Speicher, Leitwerk** (bzw. Steuerwerk, engl. *controller*), dem **Rechenwerk** (engl. *data path*) und **Ein/Ausgabe-Einheiten**.



Wo sind diese Komponenten auf PC-*Boards*?

SATA*



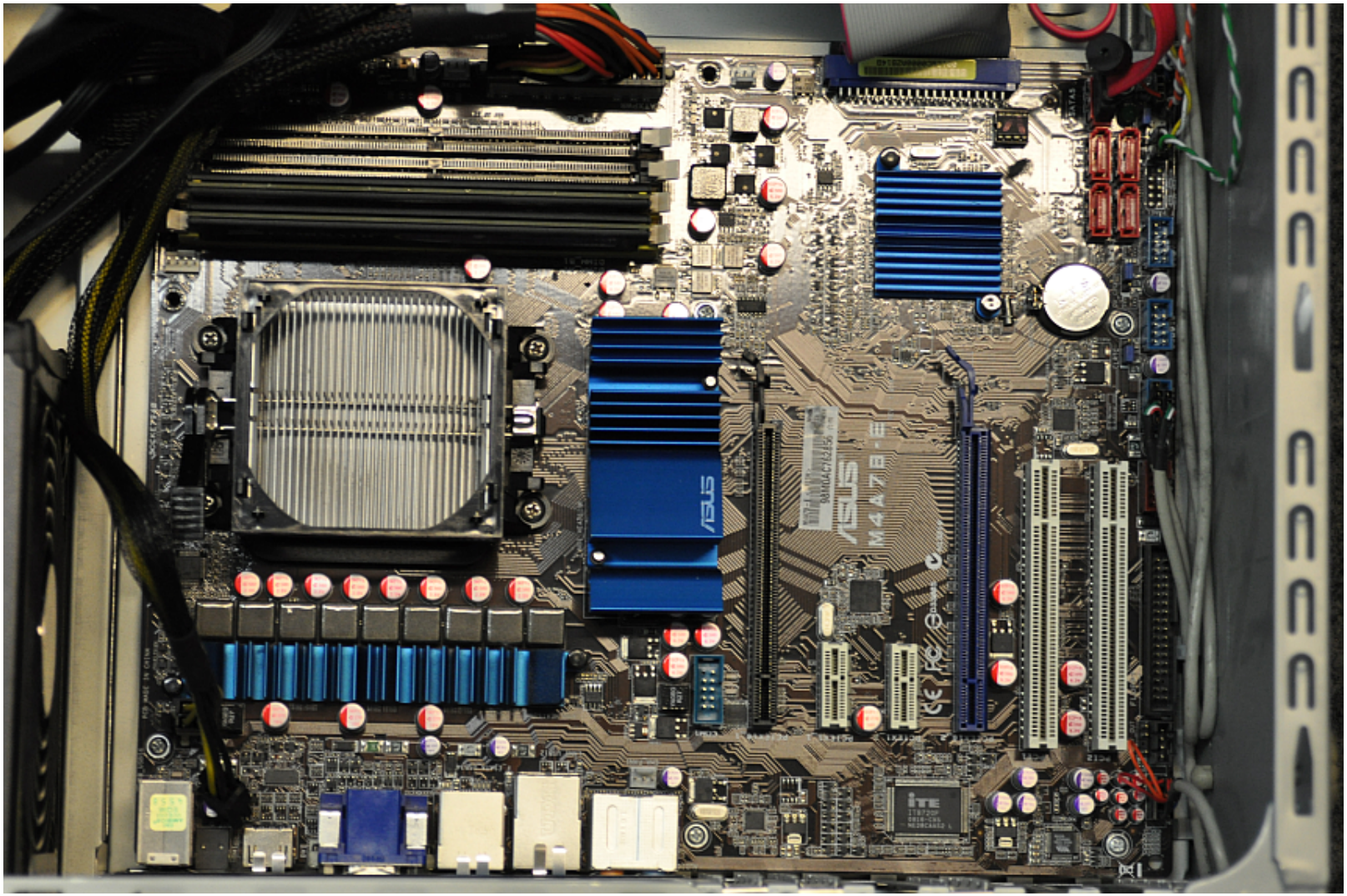
PCIe-Karte*

Speicher

Prozessor (unter Lüfter)

* Siehe Kapitel “Ein-/Ausgabe”

Wo sind diese Komponenten auf PC-*Boards* (2)?



© Photo, Informatik 12, TU Dortmund

Das Von Neumann-Modell (2)

2. Die Struktur der Anlage ist unabhängig vom bearbeiteten Problem. Die Anlage ist **speicherprogrammierbar**.
3. **Anweisungen und Operanden** (einschl. Zwischenergebnissen) werden **in demselben physikalischen Speicher** gespeichert.
4. Der Speicher wird in **Zellen gleicher Größe** geteilt. Die Zellnummern heißen **Adressen**.

5. Das Programm besteht aus einer Folge von elementaren Befehlen, die **in der Reihenfolge der Speicherung bearbeitet werden**.
6. Abweichungen von der Reihenfolge sind mit (bedingten oder unbedingten) **Sprungbefehlen** möglich.

Das Von Neumann-Modell (3)

7. Es werden **Folgen von Binärzeichen** (nachfolgend Bitvektoren genannt) verwendet, um alle Größen darzustellen.

8. Die Bitvektoren erlauben **keine explizite Angabe des repräsentierten Typs**. Aus dem Kontext muss stets klar sein, wie die Bitvektoren zu interpretieren sind.

   Alternative:

Typ	Wert

Zusammenfassung (Abschnitt)

Allgemeine Sicht auf die Befehlsschnittstelle

- Speichermodell
  - Hauptspeicher, *Endianess,* Registerspeicher
- Befehlssatz (*instruction set architecture, ISA*)
  - Befehlsgruppen, Referenzstufen, n-Adressmaschinen
  - RISC vs. CISC-Maschinen
- Interrupts
  - Ablauf, Vergleich mit Prozeduren
  - Synchrone und asynchrone Interrupts
  - Systemaufrufe
  - Wiedereintrittsfähiger (*reentrant*) Code
- Von-Neumann-Rechner (Übersicht)

Zusammenfassung (Kapitel)

- Modellierung von Rechnern auf verschiedenen Ebenen:
  - der Assemblerebene
  - der Maschinensprachenebene, …
- MIPS-Assembler- und Maschinensprachebene
  - Arithmetische, logische und Sprungbefehle
- Übersetzung einer Programmiersprache in Assembler
  - Schleifen, Prozeduren
- Allgemeine Betrachtung der sog. *instruction set architecture* (ISA)
  - Speichermodell, Maschinenbefehle, Unterbrechungen
- Charakterisierung der Von-Neumann-Maschine

1-stufige Adressierung, Referenzstufe 1 (4)

- **Register-Relative Adressierung mit Index**
  Addition zweier Register, z.B. zeigt eines auf relevanten Speicherbereich, ein zweites enthält einen Array-Index

  Beispiele:
  `ld D3,Adr(A3,D4) # D[3]:=Speicher[Adr+A[3]+D[4]] 680x0`
  IBM-370/390: nur 12 Bit für **`Adr`**, Zeiger auf relevanten Speicherbereich immer benötigt.

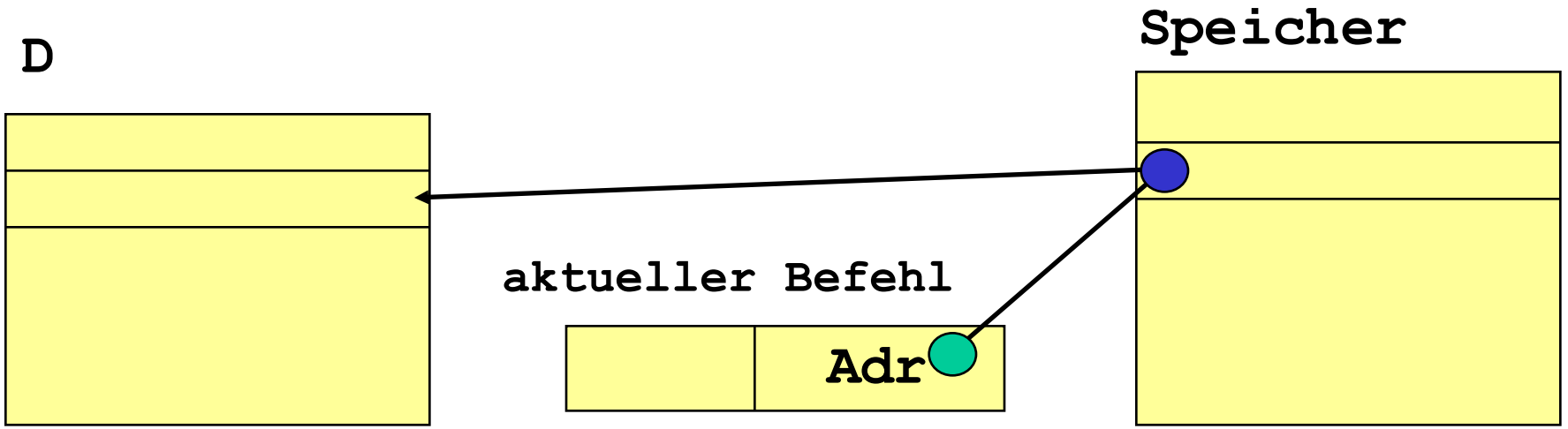
1-stufige Adressierung, Referenzstufe 1 (1)

Genau 1 Zugriff auf den Speicher

- **Direkte Adressierung, absolute Adressierung**
  Adresse ist ausschließlich im Befehlswort enthalten
  Beispiele:

Aus Befehlswort

```
lw $15,Adr      # Reg[15]:=Speicher[Adr]      MIPS
ld D3,Adr       # D[3] :=Speicher[Adr]        680x0
```



1-stufige Adressierung, Referenzstufe 1 (5)

- **Programmzähler-relative Adressierung**
  Addition des Programmzählers zu **`Adr`**

Beispiele:

```
L:bne $4,$0,Z #PC:=PC+(if Reg[4]≠0 then Z-L° else 4)
L:bra Z       #PC:=PC+(Z-L)°                    680x0
```

°Konstante wird so bestimmt, dass Sprung an Z erfolgt.