Rechnerstrukturen, Teil 2

2014/06/04

Kontext

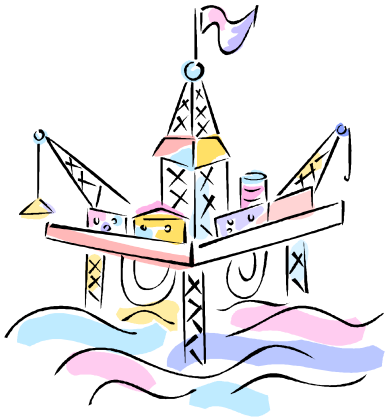
Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information
[Gesellschaft für Informatik]

Z.B. Zahlendarstellung in RS, Teil 1

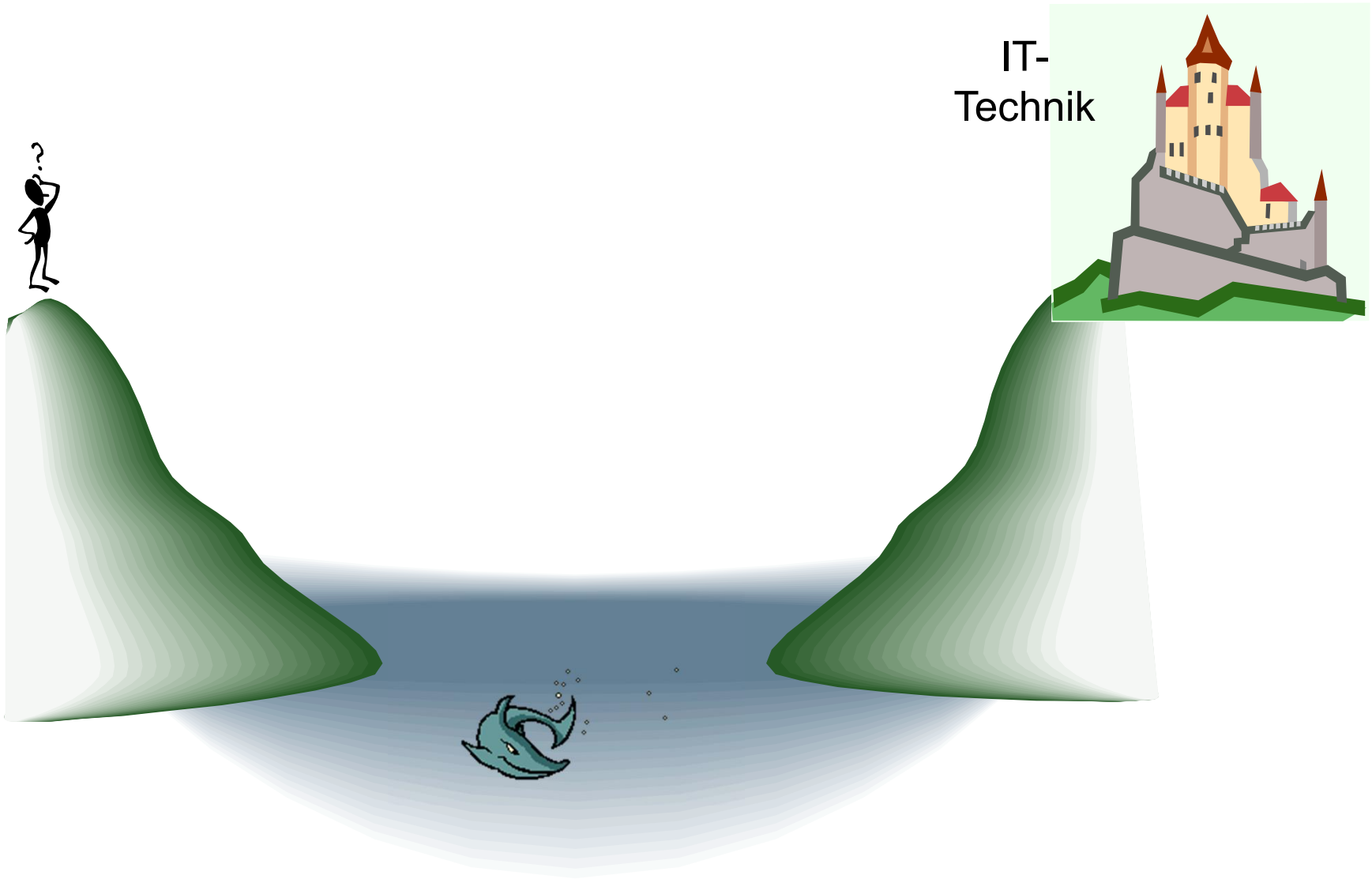
hier und jetzt

Motivation

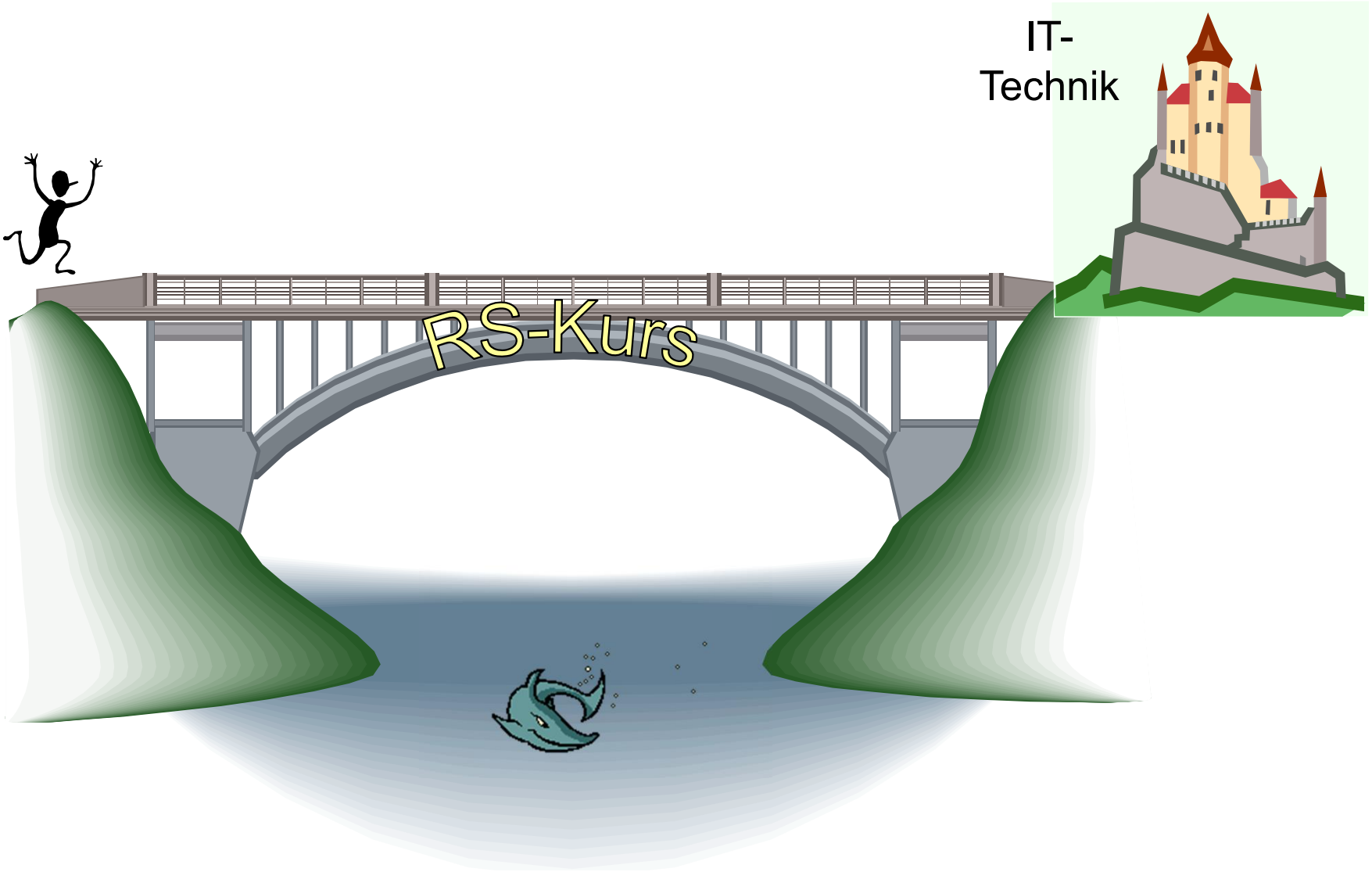
- Jede Ausführung von Programmen bedarf einer zur Ausführung fähigen Hardware.
- Wir nennen diese auch Ausführungsplattformen (***execution platforms)***.
- *Platform-based design* ist ein Ansatz für viele Anwendungen (Handys, Autos, …)
- Plattformen sind nicht immer ideal (z.B. führen Anwendungen nicht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



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

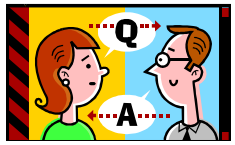
Stil des zweiten Teils der Vorlesung

Stil:

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

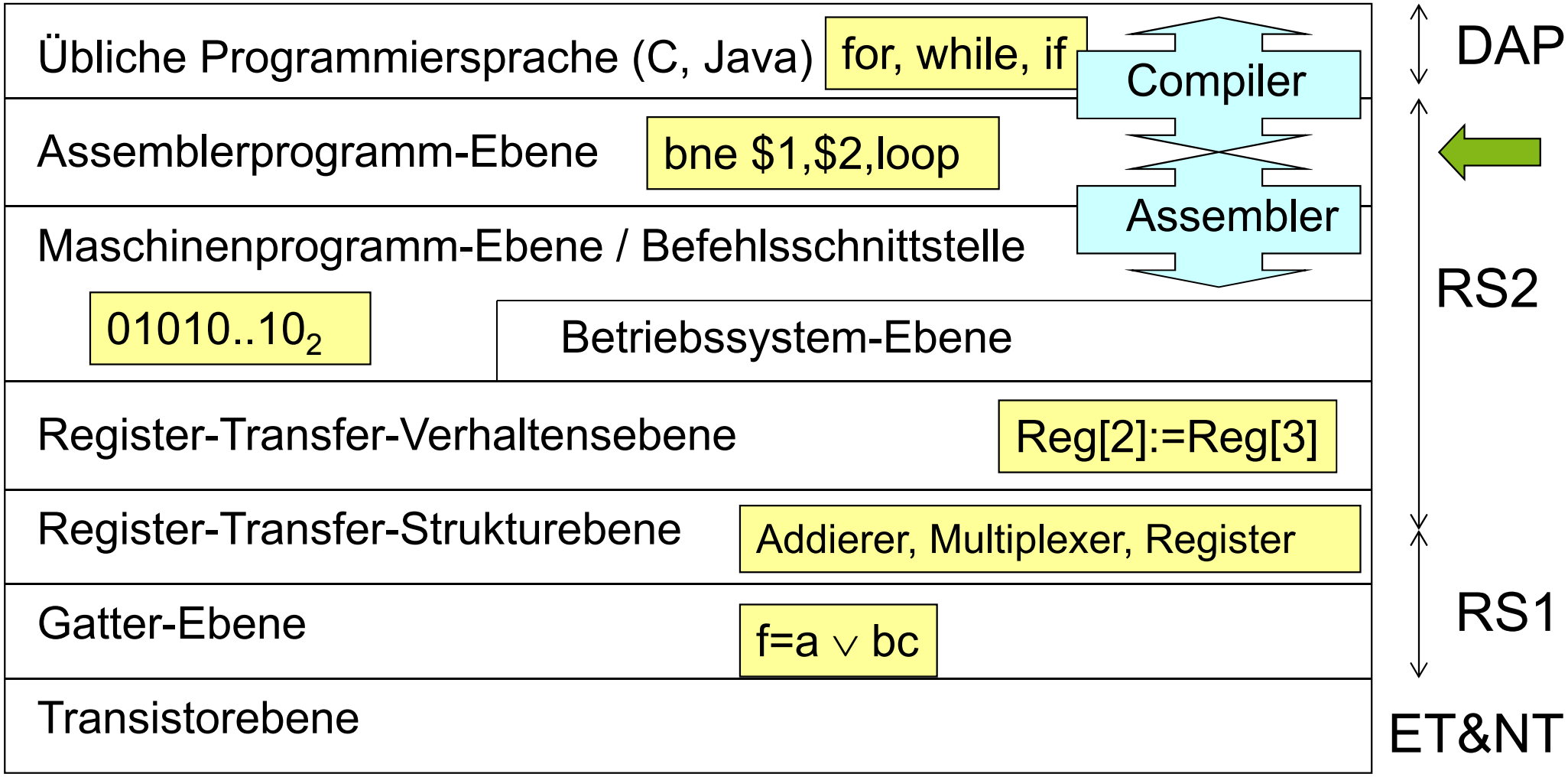
- Buch von Hennessy/Patterson: *Computer Organization: The hardware/software interface*, Morgan Kaufman, 2. Aufl. (siehe Lehrbuchsammlung) oder 3. Aufl. als grobe Leitlinie
- Weitere Bücher: siehe Literaturverzeichnis im Skript

- Soviel Interaktion, wie möglich



Rechnerarchitektur
- Einleitung -

Abstraktionsebenen:



2.2 Die Befehlsschnittstelle
2.2.1 Der MIPS-Befehlssatz

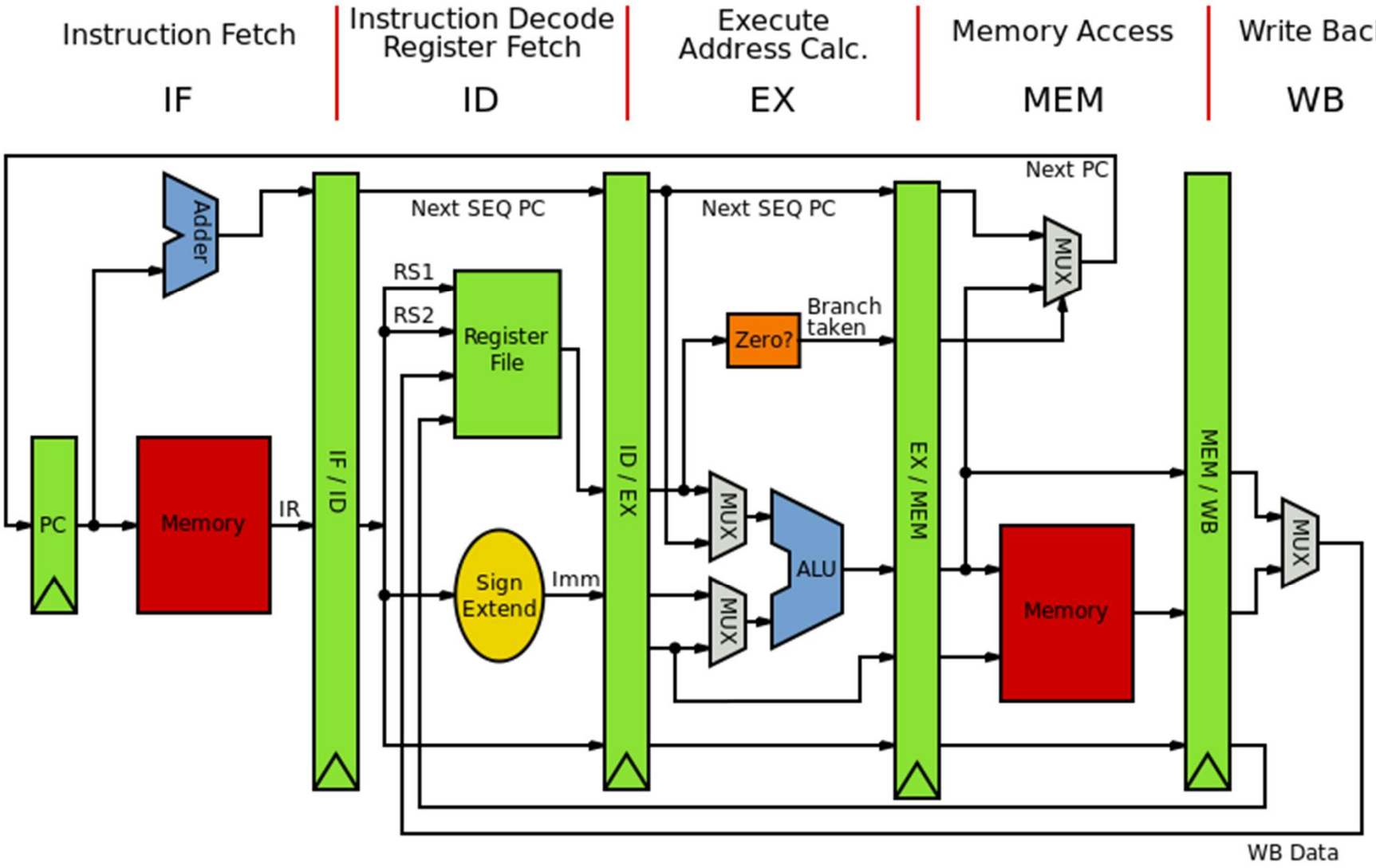
Beispiel: MIPS (~ *machine with no interlocked pipe stages*)

≠ 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

MIPS R2000
Microprocessor without interlocked pipeline stages

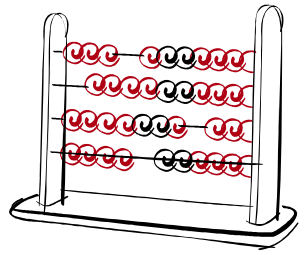


Begriffe

Assemblerprogramm-Ebene
Maschinenprogramm-Ebene / Befehlsschnittstelle

- **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,

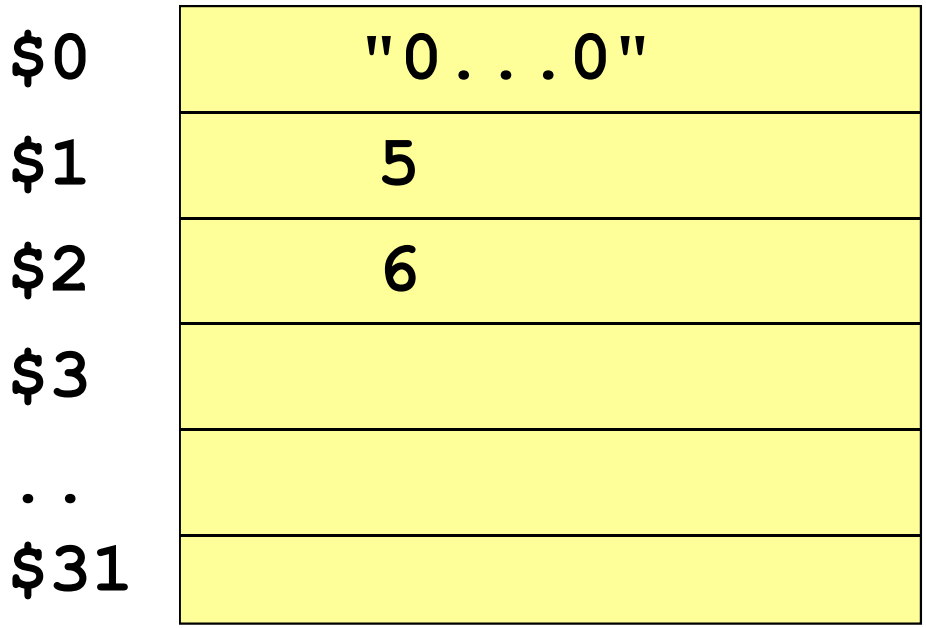
Beispiel:

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

Funktion des Additionsbefehls

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

Register („**`Reg`**“)



`$0` ist ständig = 0

`add $3,$2,$1`

Simulation des MIPS-Befehlssatzes mit MARS

Simulator MARS: www.cs.missouristate.edu/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 XP, Vista, Windows 7, MAC OS X, Linux

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

Schreiben Sie eigene kleine Programme!

Üblicherweise beziehen sich mindestens 2 von 4 Klausuraufgaben auf die MIPS-Programmierung.

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
  - 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]
```

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

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

Konstante Bitvektoren: Einschluss in ", z.B:

```
"01010100011"
```

Selektion von einzelnen Bits: Punkt + runde Klammern:

```
PC.(15:0)
```

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).

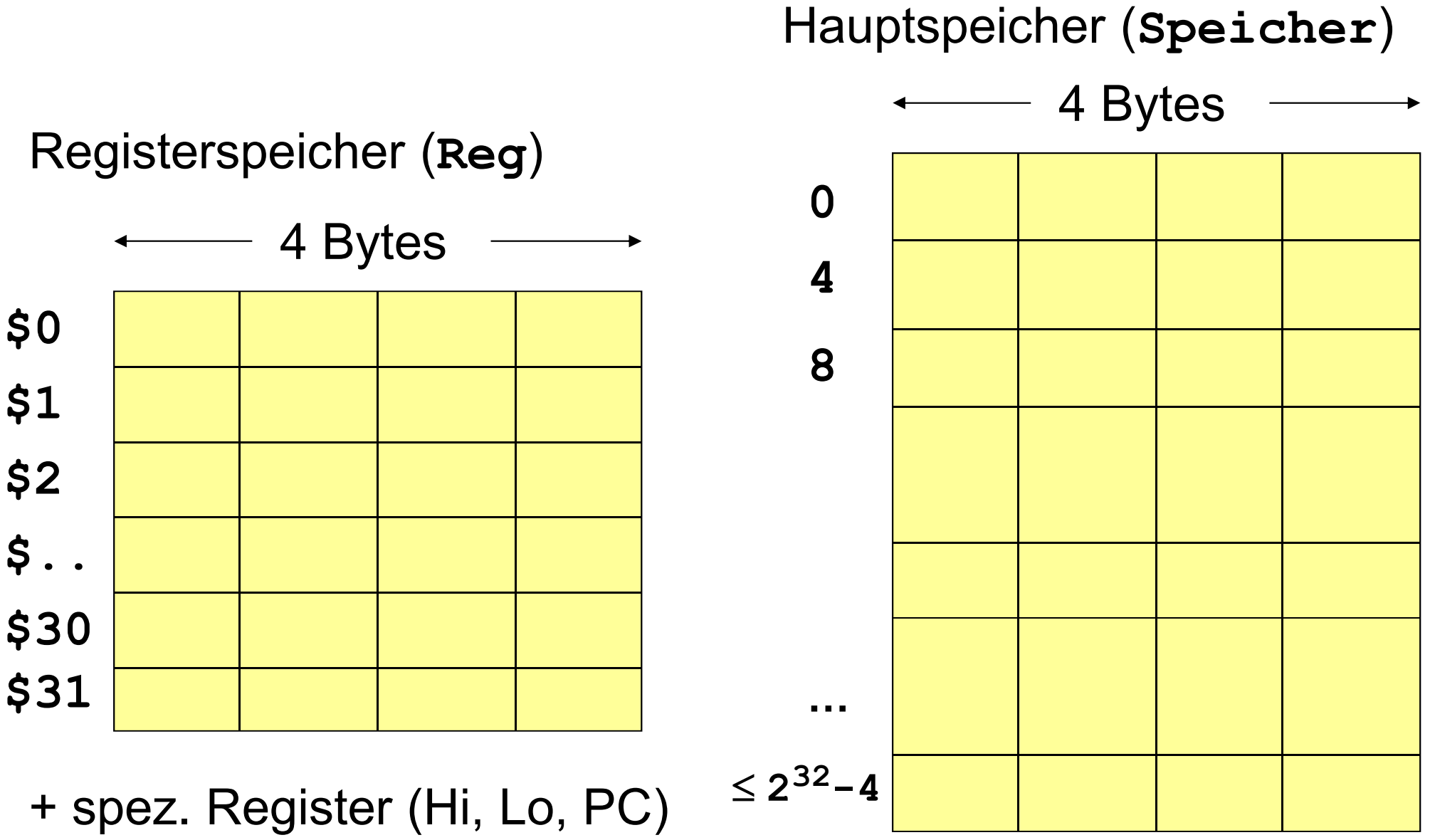
- ☞ es reicht ein Additionsbefehl zur Erzeugung der Ergebnis-Bitvektoren für beide Datentypen aus.
  Allerdings Unterschiede hinsichtlich der Behandlung von Bereichsüberschreitungen.

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).

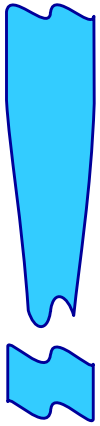
Das Speichermodell der MIPS-Architektur



Eigenschaften des Von-Neumann-Rechners (1)

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

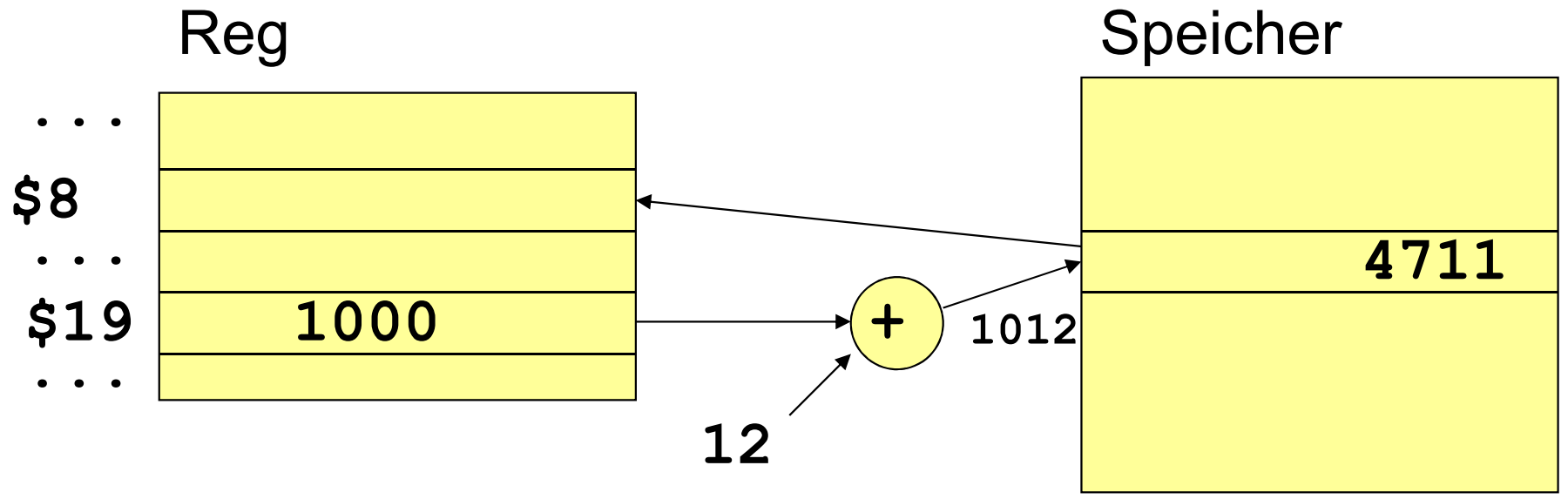
1. Einteilung des Speichers in Zellen gleicher Größe, die über **Adressen** angesprochen werden können.



Der `load word`-Befehl

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

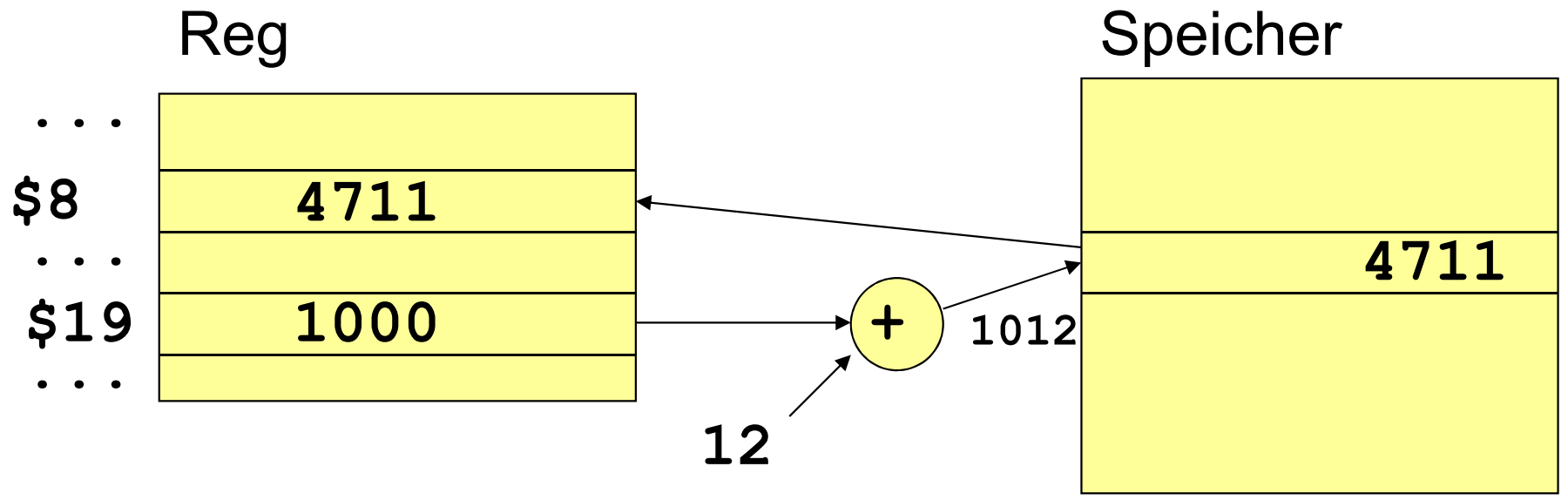
```
lw $8, 12($19)# Reg[8]:=Speicher[12+Reg[19]]
```



Der `load word`-Befehl

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

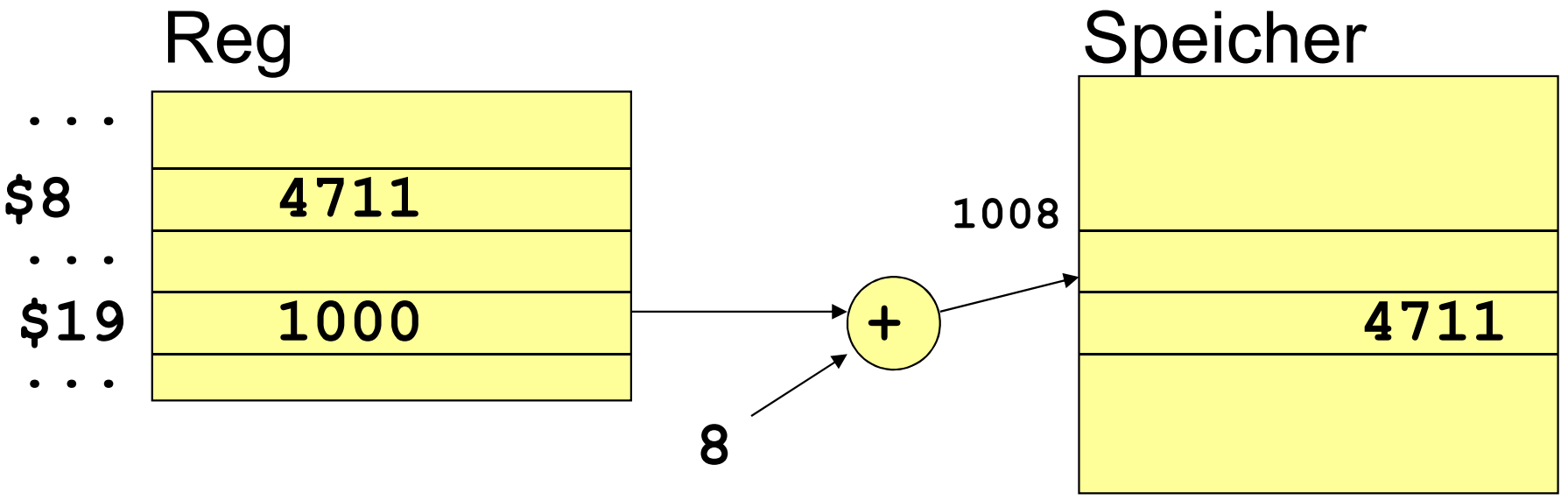
```
lw $8, 12($19)# Reg[8]:=Speicher[12+Reg[19]]
```



Der `store word`-Befehl

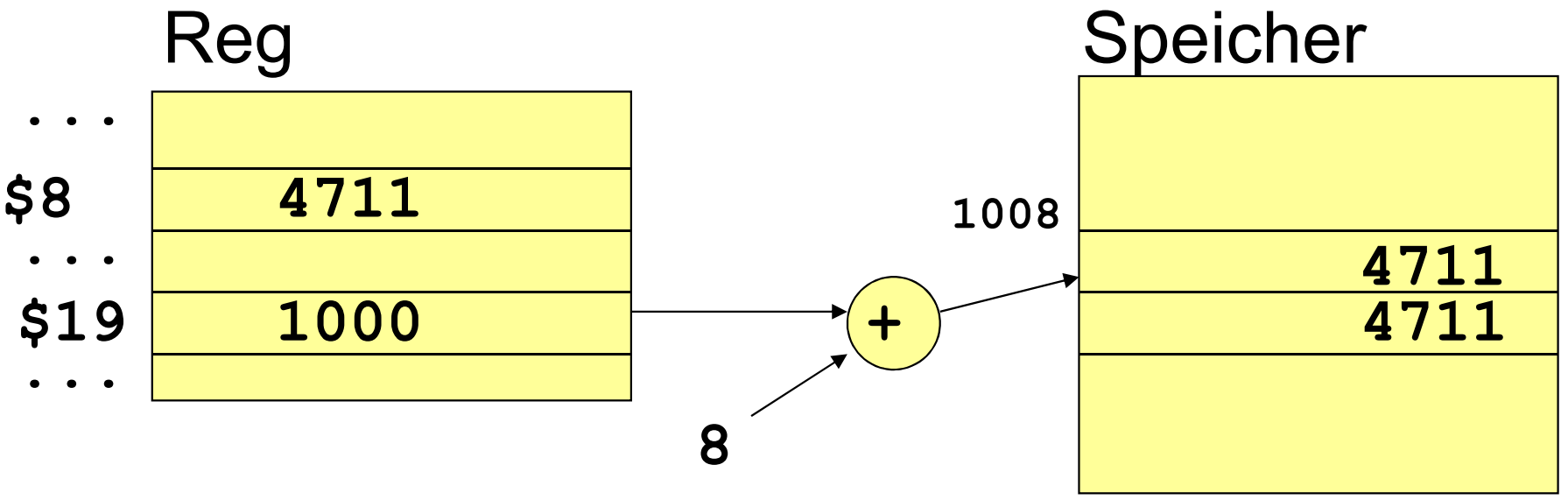
Allgemeine Form:
`sw quel, offset(reg)` mit **`quel, reg`** $\in$ **`$0..$31`**,
`offset:` Konstante $\in -2^{15} .. 2^{15}-1$, deren Wert beim Laden des Programms bekannt sein muss bzw. deren Bezeichner.
Beispiel:

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



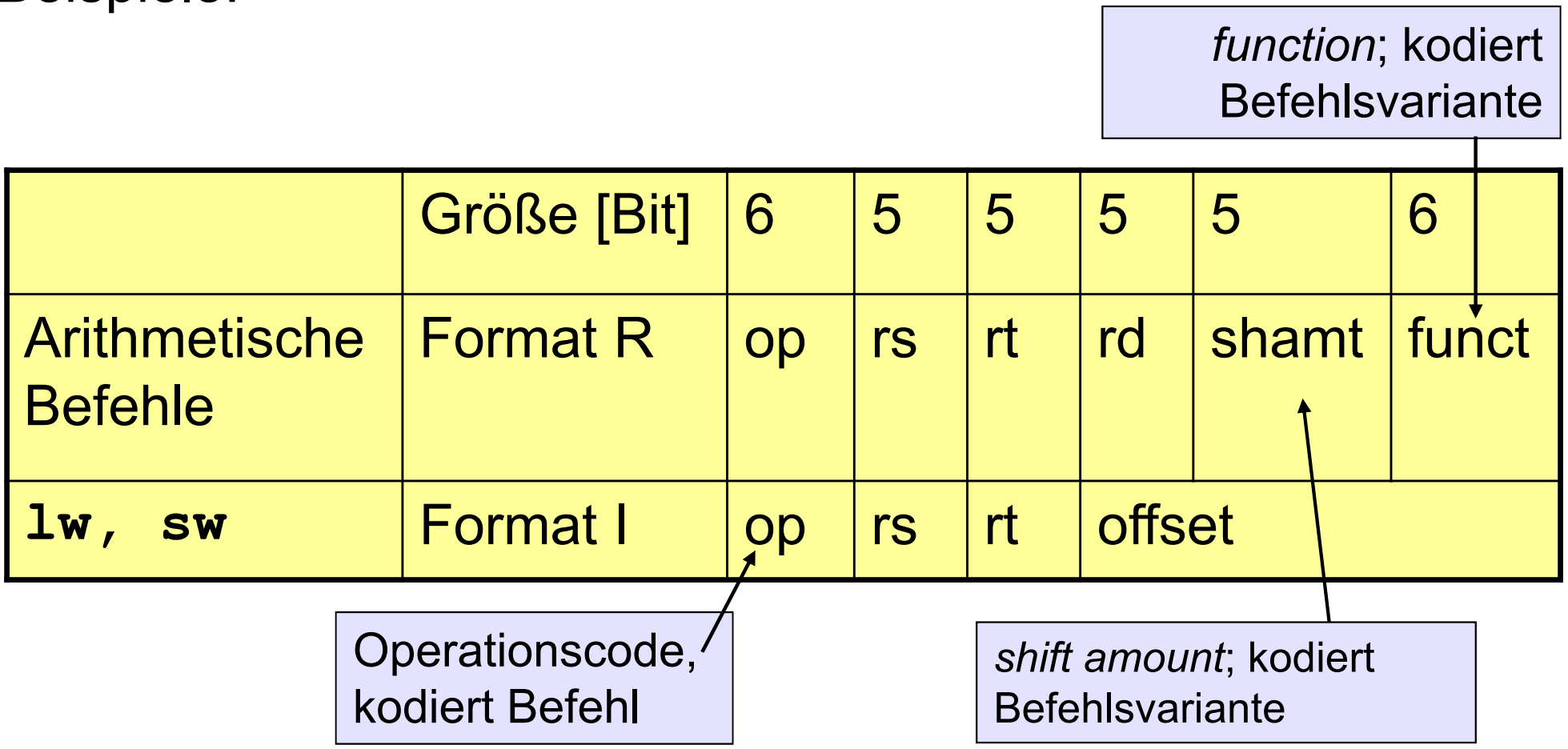
Der `store word`-Befehl

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:



	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		

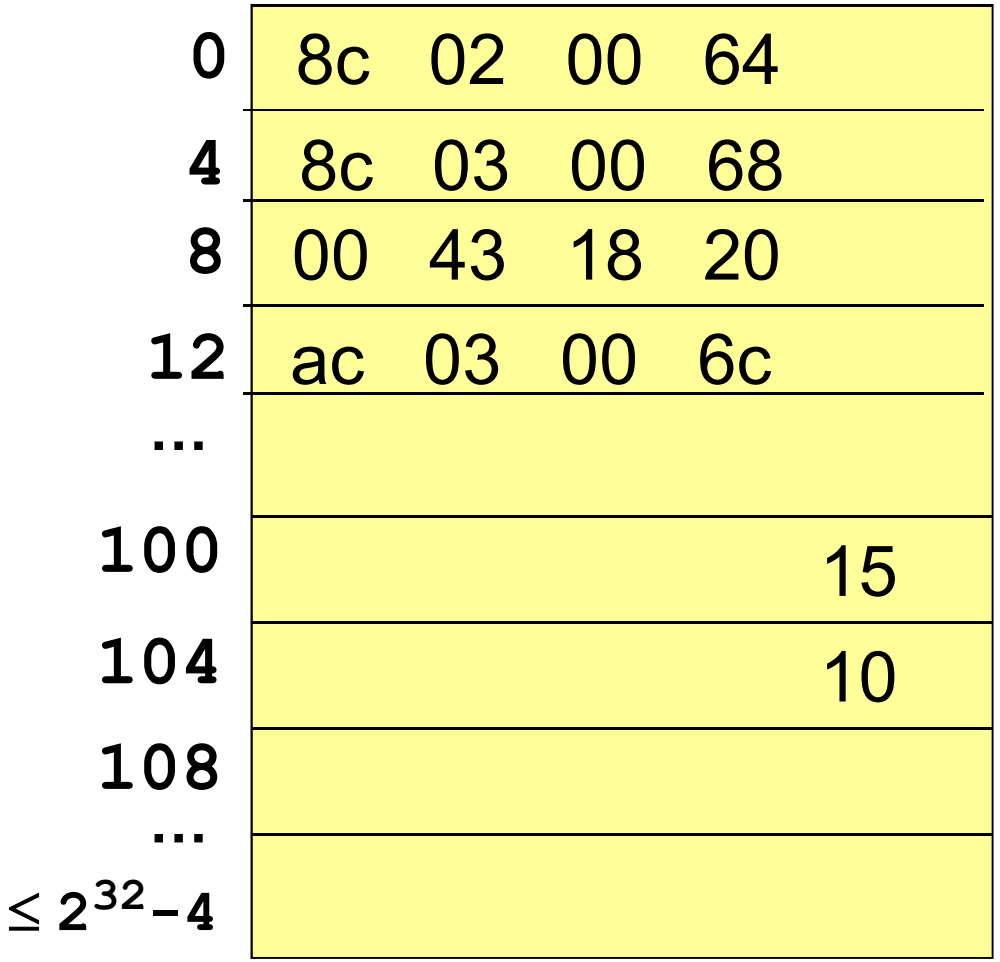
Speicherung von Befehlen im Rechner

Assemblerprogramm

```
lw   $2,100($0)
lw   $3,104($0)
add $3,$2,$3
sw   $3,108($0)
```

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

Hauptspeicher (**`Speicher`**)

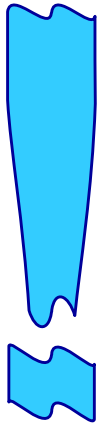


Eigenschaften des Von-Neumann-Rechners (2)

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**)

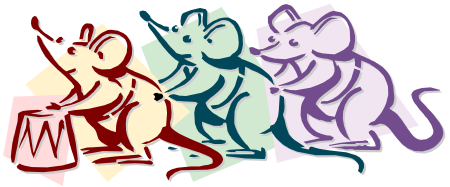
0	8c 02 00 64	lw $2,100($0)
4	8c 03 00 68	lw $3,104($0)
8	00 43 18 20	add $3,$2,$3
12	9c 03 00 6c	sw $3,108($0)
...		
100	15	
104	10	
108	25	
...		
$\leq 2^{32}-4$		

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

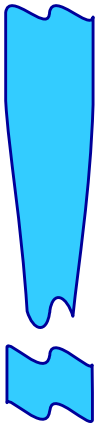
Eigenschaften des Von-Neumann-Rechners (3)

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

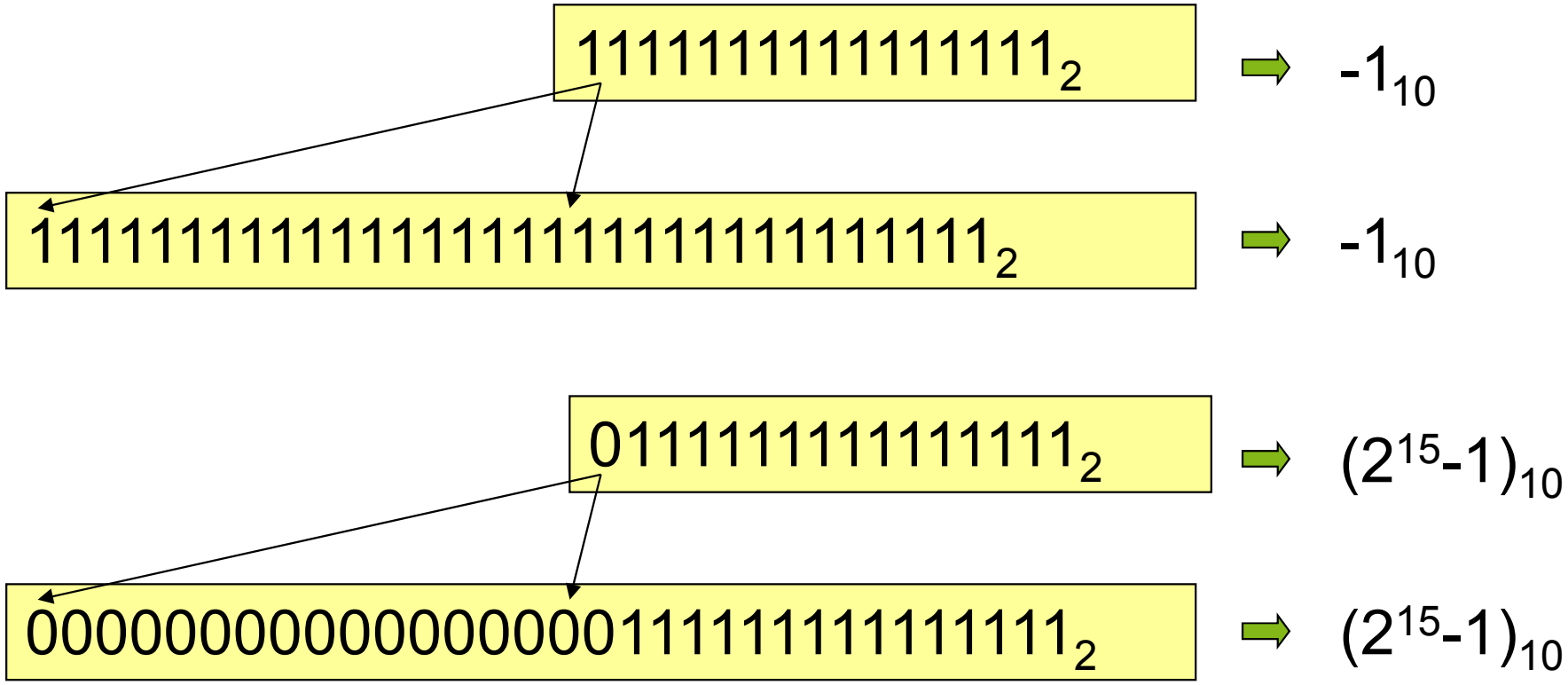
Darstellung von Befehlen im Rechner (Wdh.)

	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

Beweis der Korrektheit von `sign_ext`

Bitvektor a=(a_{n-1}, …,a_0) repräsentiert bei 2k-Kodierung Zahl

$$\text{int}(a) = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Sei nun a_{n-1}='0'. Dann ist

$$\text{int}(\text{sign\_ext}(a,m)) = \text{int}("\overbrace{00..0}^{m-n+1}"\&(a_{n-2},..a_0)) = \sum_{i=0}^{n-2} a_i 2^i = \text{int}(a)$$

- Sei nun a_{n-1}='1'. Dann ist

$$\text{int}(\text{sign\_ext}(a,m)) = \text{int}("\overbrace{11..1}^{m-n+1}"\&(a_{n-2},..a_0)) = -2^{m-1} + \sum_{i=n-1}^{m-2} 2^i + \sum_{i=0}^{n-2} a_i 2^i$$

Wegen $\sum_{i=n-1}^{m-2} 2^i + 2^{n-1} = 2^{m-1}$ folgt $-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i = -2^{n-1}$, also auch

$$\text{int}(\text{sign\_ext}(a,m)) = -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = \text{int}(a)$$ q.e.d.

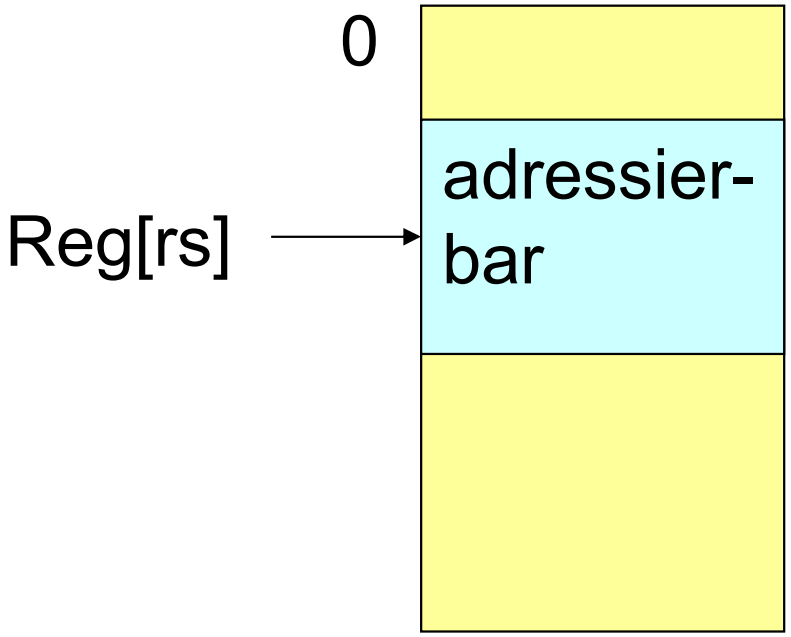
Auswirkung der Nutzung von `sign_ext` bei der Adressierung

Präzisere Beschreibung der Bedeutung des sw-Befehls:

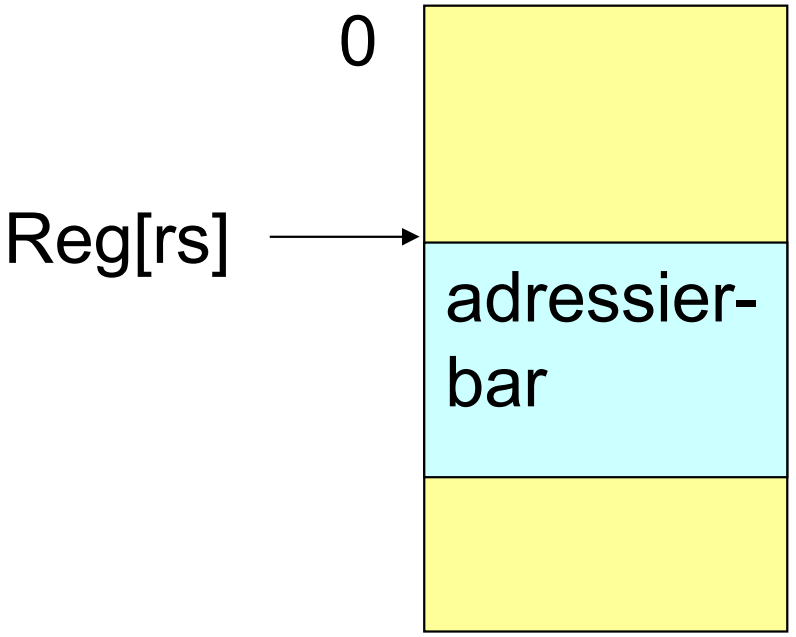
`sw rt, offset(rs) #`

`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)`

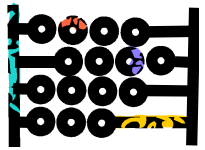
- *add immediate unsigned*-Befehl

  `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:

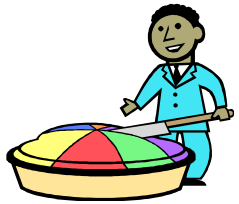
```
mfhi, $3 # Reg[3]:=Hi
mflo, $3 # Reg[3]:=Lo
```

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,$3 #`** für natürliche Zahlen (*unsigned integers*)
  `# Lo:=Reg[2]/`$_{\mathbf{u}}$ **`Reg[3]; Hi:=Reg[2] mod Reg[3]`**

Logische Befehle

Beispiel	Bedeutung	Kommentar
`and $4,$3,$2`	`Reg[4]:=Reg[3] ∧ Reg[2]`	und
`or $4,$3,$2`	`Reg[4]:=Reg[3] ∨ Reg[2]`	oder
`andi $4,$3,100`	`Reg[4]:=Reg[3] ∧ 100` (zero_ext( )	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

Laden von Konstanten

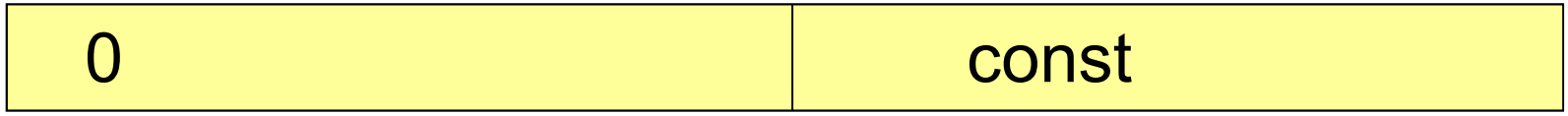
Wie kann man 32-Bit-Konstanten in Register laden?

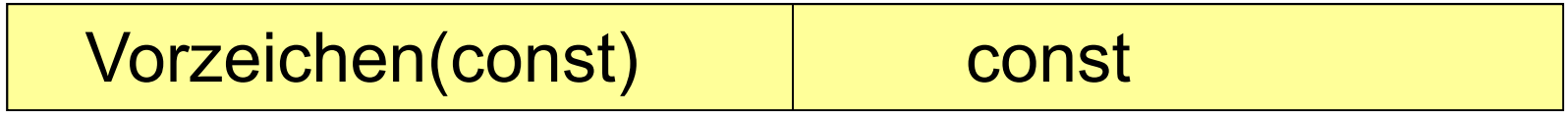
- Direktoperanden für das untere Halbwort:

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

- Für den Sonderfall s=$0:

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





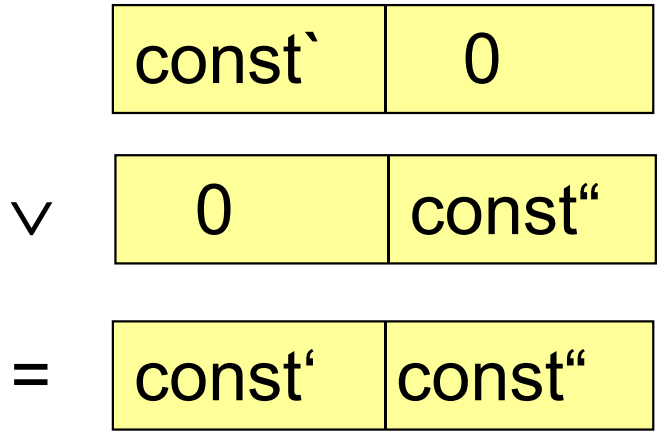
Laden von Konstanten (2)

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

const	0

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.

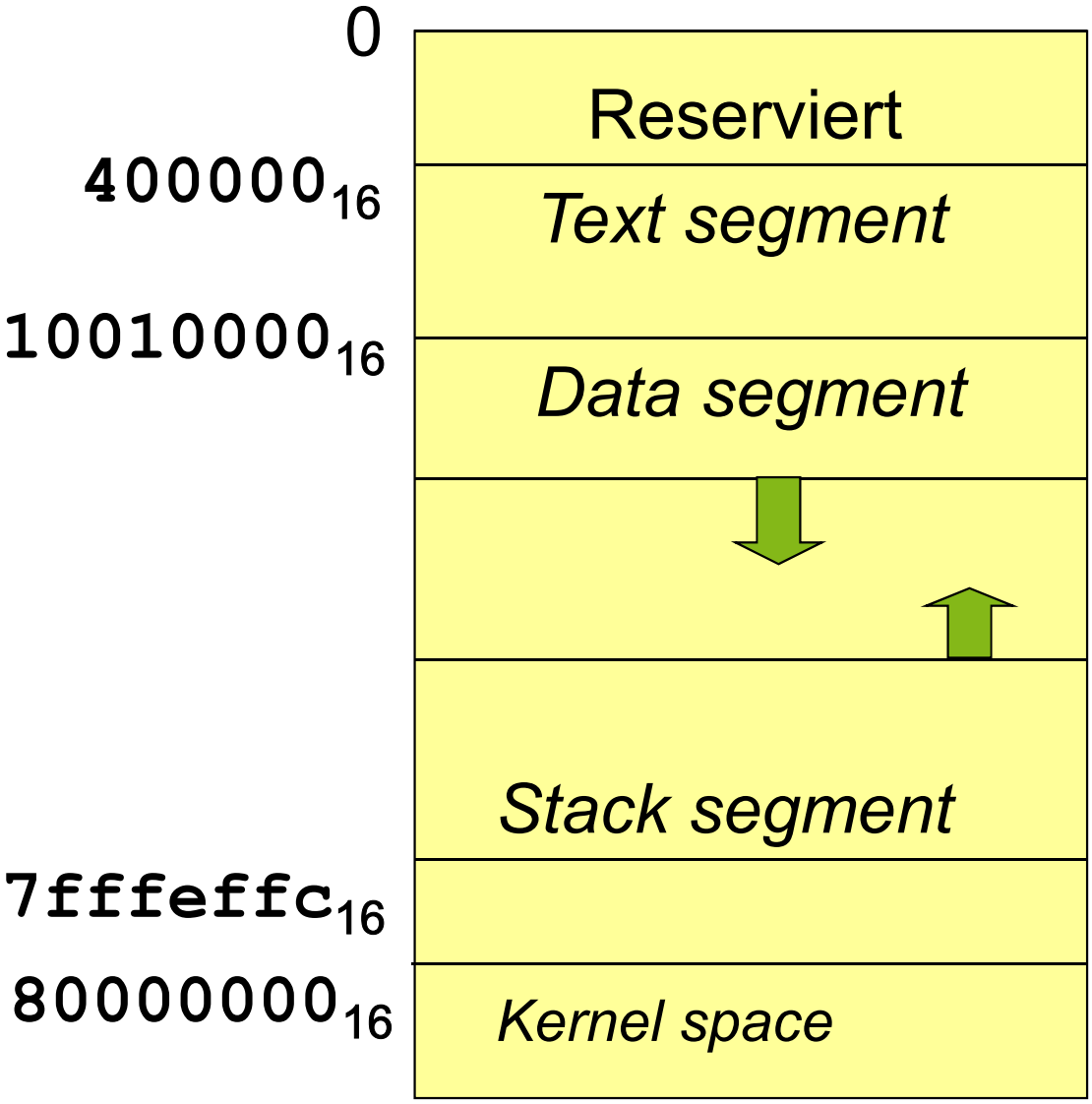
Dies ist mit den bislang vorgestellten Befehlen zwar möglich, der Lesbarkeit wegen wird ein eigener Befehl eingeführt.

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



Befehle

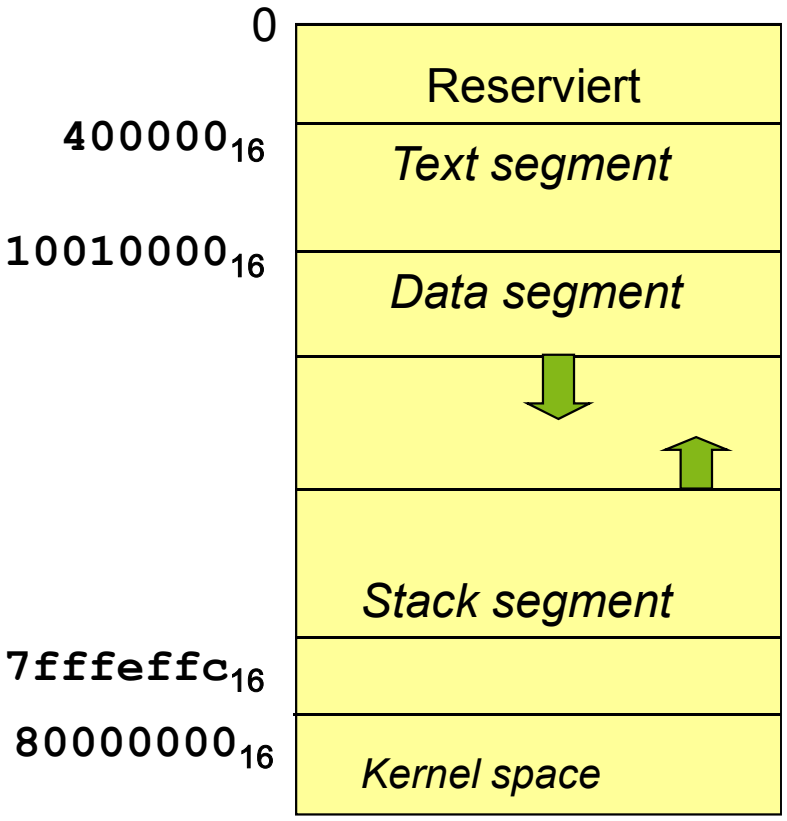
Daten bekannter Größe+weitere Daten

Dynamisch erzeugte Daten, Auf- und Abbau bei Aufruf von Prozeduren/Unterprogrammen

Adresse Ein-/ Ausgabegeräte

Betriebssystem

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)
```

MARS erzeugt 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)
```

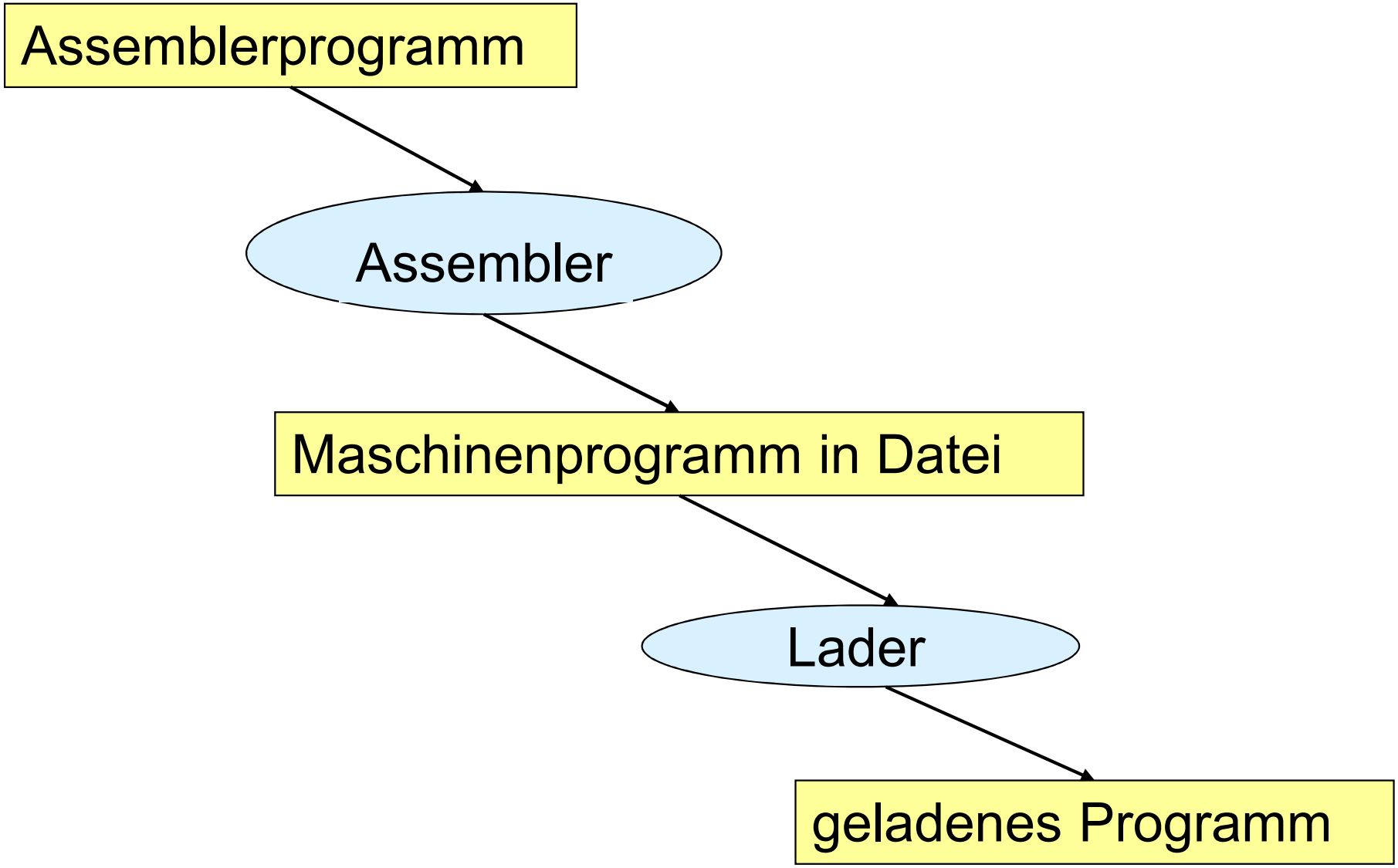
Der 2. Befehl entfällt wenn ($b) nicht vorhanden ist, aber nicht bei b=0 (!)

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