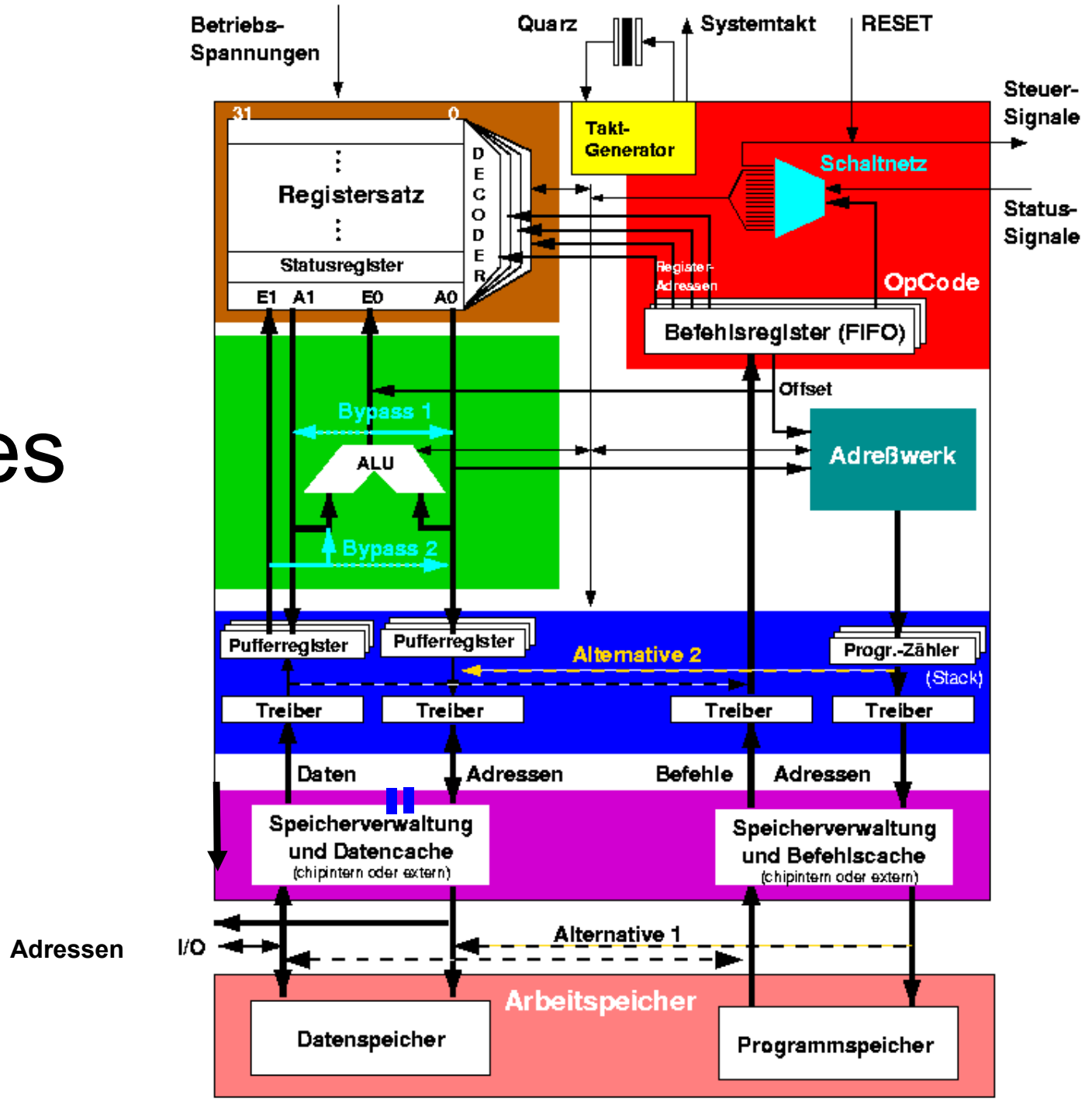
2. Übung

Assemblerprogrammierung mit dem MIPS-Simulator MARS

- Der MARS-Simulator (MIPS-32-Prozessor)
  - Literatur: Hennessy & Patterson (Anhang A auf der TI-Homepage)
- MIPS-Programmiermodell
  - Aufbau eines MIPS-Prozessors
  - Registersatz
  - Speicheraufteilung
- MIPS-Assemblerprogrammierung
  - Syntax der MIPS-Assemblersprache
  - Assemblerdirektiven
  - Adressierungsarten
  - Datenformate
  - Befehlsformate & Befehlssatz

Aufbau eines RISC-Prozessors



Warum MIPS?

MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages)

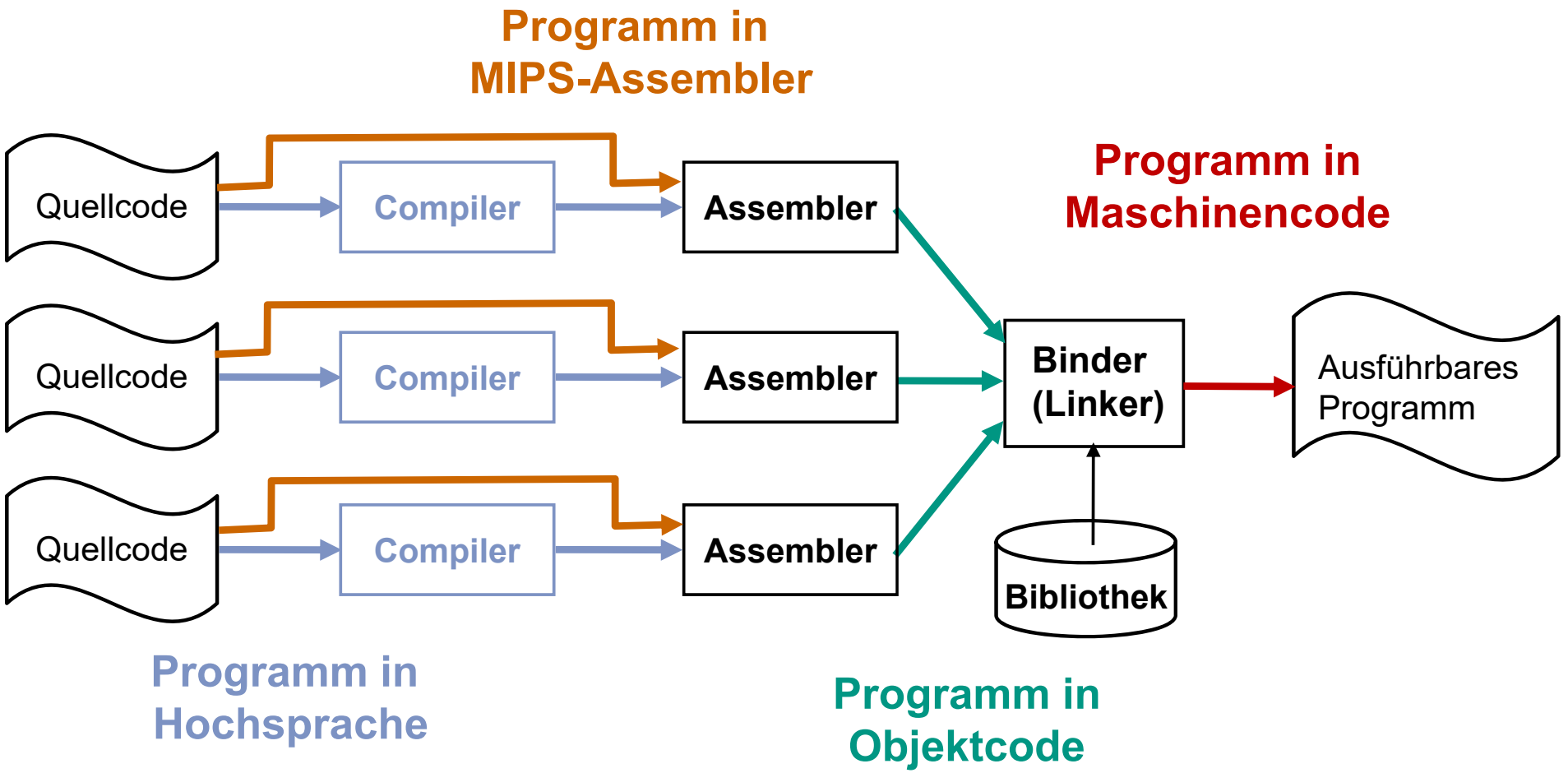
≠

MIPS (*million instructions per second*)

- MIPS Prozessoren sind sehr klar strukturierte RISC-Prozessoren
- Sauberer und klarer Befehlssatz
- Basis der richtungsweisenden Bücher von Hennessy/Patterson
- MIPS außerhalb von PCs weit verbreitet: Windows CE Geräte, Cisco-Router, SGI‘s und Videospiel-Konsolen: Nintendo 64, Sony PlayStation, PlayStation 2, PlayStation Portable
- Simulator (MARS) verfügbar
  - Keine Gefährdung des laufenden Systems
  - Bessere Interaktionsmöglichkeiten
  - MARS ist Java-basiert, läuft unter allen gängigen Betriebssystemen (Linux, Windows, Mac OS)

Der MARS-Simulator

MARS (MIPS Assembly and Runtime Simulator) ist Assembler, Linker und Debugger in einem Programm.



Der MARS-Simulator

- **Assembler:**
  Programm, das einen Quellcode in Assemblersprache
  in eindeutiger Weise in Maschinensprache übersetzt.
- **Objektcode:** Repräsentation eines Maschinenprogramms, in dem noch ungelöste Referenzen auf externe Unterprogramme oder Speicherbereiche enthalten sind.
- Zusätzlich können im Objektcode Informationen
  enthalten sein, die die Fehlersuche mit einem **Debugger** ermöglichen.
- **Binder (Linker):** Programm, das die ungelösten Referenzen mehrerer Objektcode-Module auflöst und sie zu einem ausführbaren Programm verbindet.

Installation und Benutzung

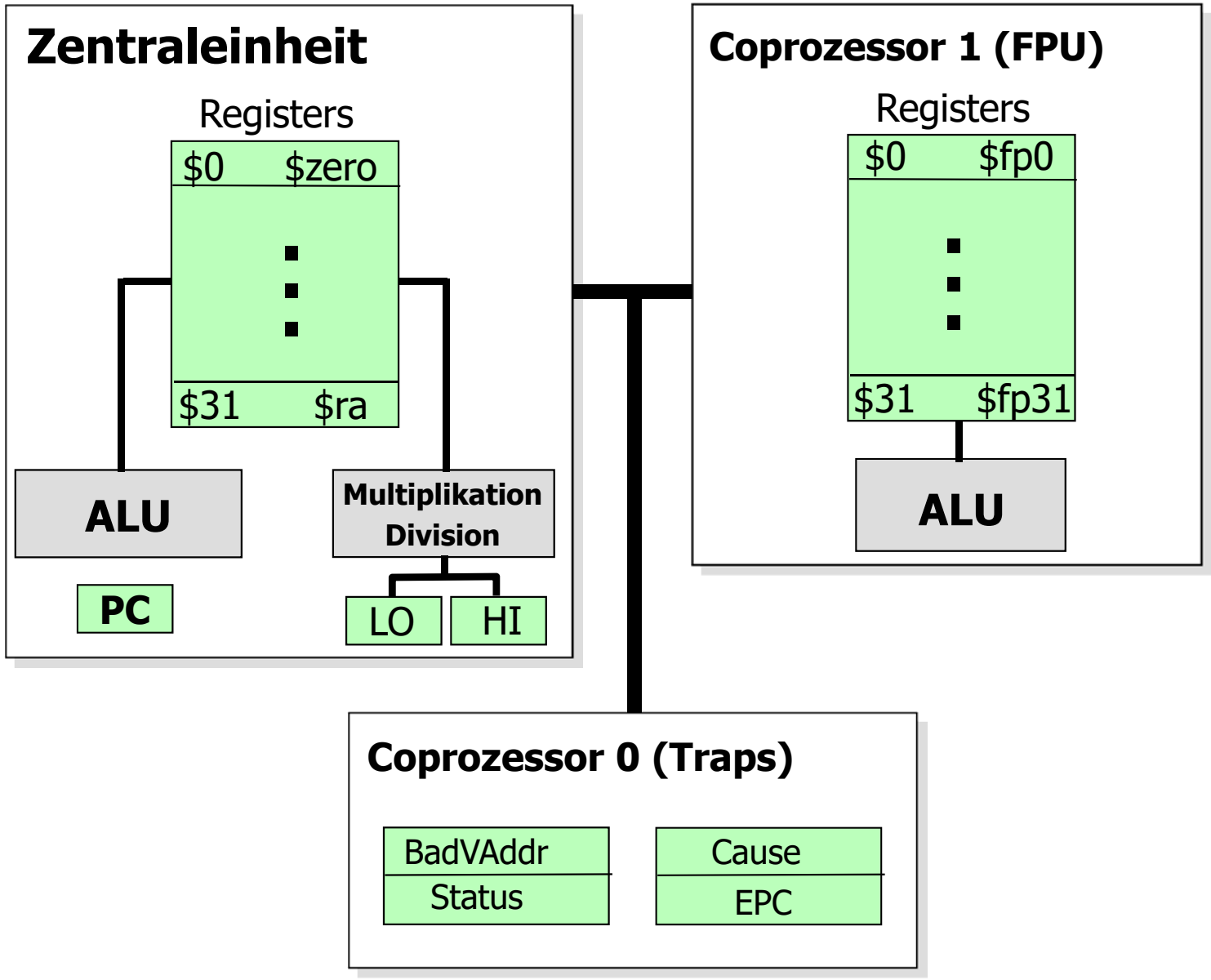
- Weitere Materialien zu MIPS sind abrufbar unter:

  http://ti.ira.uka.de/TI-2/Mips/Mips.php

- Der MIPS-Simulator „MARS" unterliegt der MIT-Lizenz und ist somit Open Source. Die neueste Version ist erhältlich unter:

  http://courses.missouristate.edu/kenvollmar/mars/

Aufbau des MIPS-Prozessors



Koprozessoren

Der MIPS-Prozessor besitzt zwei Koprozessoren:

- **Coprozessor 0 (Traps):**
  Register enthalten Informationen über den
  Prozessorstatus und die Ursachen von Unterbrechungen und Ausnahmen
- **Coprozessor 1 (FPU) für Gleitkomma-Arithmetik:**
  Enthält 32 allgemein verwendbare 32-Bit-Gleitkomma-Register

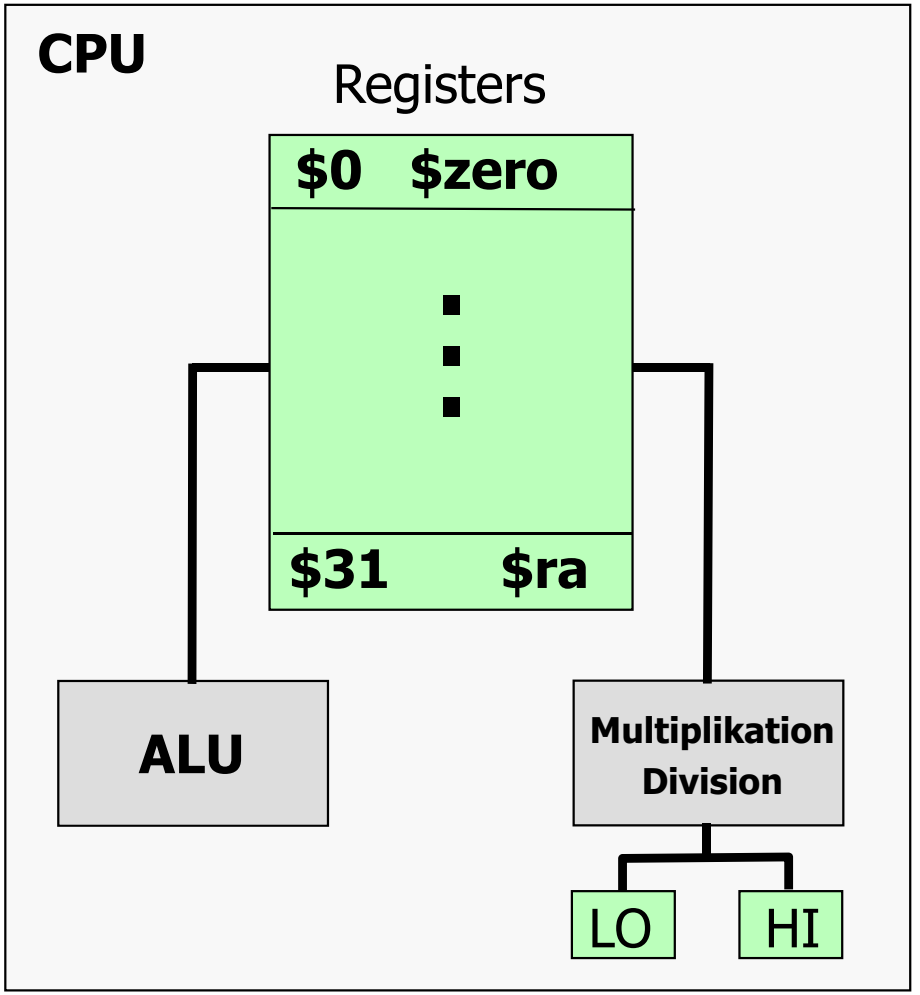
Registersatz

MIPS ist eine Lade/Speicher-Architektur:

- Speicherzugriffe nur über Lade- und Speicher-Befehle.
- Berechnungen erfolgen nur auf Registern

Der MIPS-Prozessor ist eine typische **Register-Register-Maschine** mit **32** allgemein verwendbaren Registern.

Sie sind durch ein vorangestelltes **$**-Zeichen gekennzeichnet und deren Verwendung ist zum Teil durch Konvention festgelegt.



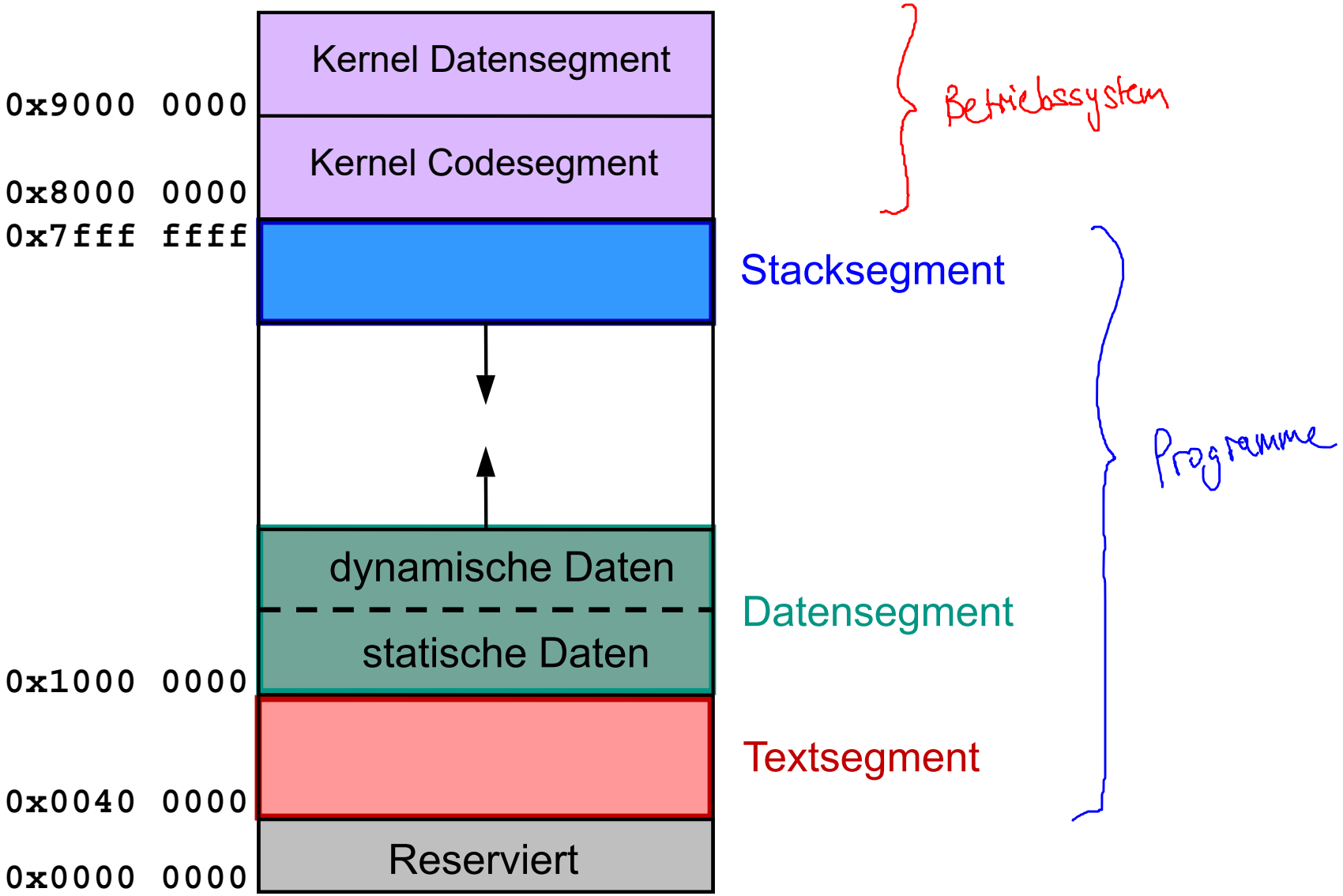
Registersatz

Name	Nr.	Verwendung
$zero	$0	Konstante mit dem Wert 0 Kann nicht verändert werden.
$at	$1	Reserviert für den Assembler (temporäres Register zur Erzeugung von Pseudobefehlen). Darf vom Programmierer **nicht** verwendet werden.
$v0 – $v1	$2 – $3	Rückgabe von Funktionswerten
$a0 – $a3	$4 – $7	Übergabe der ersten vier Argumente an Unterprogramme oder Funktionen verwendet. Weitere Argumente werden auf dem Stack übergeben.
$t0 – $t7 $t8 – $t9	$8 – $15 $24 – $25	Register für temporäre Variablen. Sie müssen ggf. **vor Unterprogrammaufruf** gesichert werden.
$s0 – $s7	$16 – $23	Register für langlebige Variablen. Sie müssen **vom Unterprogramm** gesichert werden.

Registersatz

Name	Nr.	Verwendung
$k0 – $k1	$26 – $27	Reserviert für das Betriebssystem. Dürfen vom Programmierer nicht verwendet werden.
$gp	$28	Beinhaltet einen Zeiger auf die Mitte im statischen Datensegment und wird zum schnellen Laden und Speichern globaler Daten verwendet.
$sp	$29	Stack-Zeiger: Verweist auf die erste freie Speicheradresse des Stacks
$fp	$30	Rahmen-Zeiger: Für die temporäre Allokierung von Speicherplatz beim Aufruf von Unterprogrammen
$ra	$31	Enthält die Rücksprungadresse beim Unterprogrammaufruf.
PC	–	Befehlszähler
HI, LO	–	64-Bit-Resultat einer Multiplikation von Integer-Zahlen bzw. Quotient und Rest einer Integer-Division

Speicheraufteilung



Speicheraufteilung

- **Textsegment:** beinhaltet den ausführbaren Maschinencode
- **Datensegment:** beinhaltet statische und dynamische Daten:
  - Speicherbereiche für statische Daten werden vom *Assembler* allokiert. (*In C: globale Variablen)*
  - Speicherbereiche für dynamische Daten werden vom *Programm* allokiert. *(In C: void *malloc(size))*
- **Stacksegment:** beinhaltet lokale Daten und Rücksprungadressen für Unterprogramme
- **Kernelsegmente:** beinhalten betriebssystemeigene Daten und Prozeduren

Syntax der MIPS-Assemblersprache

Ein Assemblerprogramm besteht aus:

- Assemblerdirektiven
- Marken (Labels)
- Maschinenbefehlen (und Pseudobefehlen, Makros)
- Kommentaren

Ein **Bezeichner** ist eine Folge von alphanumerischen Zeichen, Unterstrichen (_) und Punkten (.), die nicht mit einer Ziffer beginnen. Opcodes für Maschinenbefehle und Assemblerdirektiven dürfen nicht als Bezeichner verwendet werden.

Eine **Marke** ist ein Bezeichner, der am Beginn einer Zeile steht und mit einem Doppelpunkt (:) abgeschlossen wird. Eine Marke steht für eine symbolische Referenz auf eine Speicheradresse. Die Marke *main* ist für das Hauptprogramm reserviert.

Syntax der MIPS-Assemblersprache

Strings innerhalb des Quelltextes werden in Hochkomma (") eingeschlossen. Spezielle Zeichen werden entsprechend der C-Konvention dargestellt:

Neue Zeile	\n
Tabulator	\t
Hochkomma	\"

Kommentare beginnen mit einem #-Zeichen und erstrecken sich bis zum Zeilenende, Keine mehrzeiligen Kommentare

Grundsätzlich gilt:

Jede Befehlszeile sollte kommentiert werden!

MIPS-Assemblerdirektiven

Assemblerdirektiven beginnen mit einem Punkt (.)

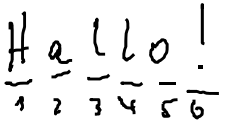
`.align n` .align 2 $2^2=4$

Die folgenden Daten sollen an der nächstmöglichen Adresse gespeichert werden, die durch $\mathbf{2^n}$ teilbar ist

`.space n`

Allokiert **n** Bytes im aktuellen (Daten-)Segment

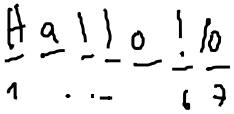
`.ascii string` label2: .ascii "Hallo!"



Allokiert Speicher und legt den String **`string`** im Speicher ab

`.asciiz string`

Allokiert Speicher und legt den String **`string`** im Speicher ab und hängt ein Null-Byte an



MIPS-Assemblerdirektiven

`.byte` $b_1,...,b_n$ data1: .byte 1,2,3,4

Allokiert Speicher und legt **n** Daten vom Typ Byte (8 Bit) im Speicher ab

`.half` $h_1,...,h_n$

Allokiert Speicher und legt **n** Daten vom Typ Halbwort (16 Bit) im Speicher ab

`.word` $w_1,...,w_n$

Allokiert Speicher und legt **n** Daten vom Typ Wort (32 Bit) im Speicher ab

`.float` $f_1,...,f_n$

Allokiert Speicher und legt **n** Fließkommazahlen (einfache Genauigkeit) im Speicher ab

MIPS-Assemblerdirektiven

`.double` $d_1,...,d_n$

Allokiert Speicher und legt **n** Fließkommazahlen (doppelte Genauigkeit) im Speicher ab

`.globl symbol`

Deklariert die Marke **`symbol`** als globales Symbol (kann von anderen Dateien referenziert werden)

`.extern symbol size`

Deklariert das Datum, welches bei der Marke **`symbol`** gespeichert ist, als globales Symbol der Größe **`size`** Bytes. Das Datum wird derart im Datensegment gespeichert, so dass ein effizienter Zugriff mittels des Register **`$gp`** möglich wird

MIPS-Assemblerdirektiven

`.(k)data <addr>`

.data .kdata

Folgende Daten sollen im (Kernel-)Datensegment gespeichert werden. Optional kann eine Adresse `<addr>` angegeben werden. Der Assembler beginnt ab Adresse $1001\ 0000_{16}$ Daten im Datensegment abzulegen

`.(k)text <addr>`

.text .ktext

Folgende Daten sollen im (Kernel-)Textsegment gespeichert werden. Optional kann eine Adresse `<addr>` angegeben werden

`.set (noat|at)`

Warnung des Assemblers über die Benutzung des `$at`-Registers ein- oder ausschalten

Beispiel: MIPS-Assemblerdirektiven

```
            .data                   # Es folgen Daten, die
                                    # im Datensegment stehen.
Note:       .asciiz "Note"          # String
PI:         .float 3.1415           # Konstante
x:          .space 4                # Allokiere 4 Bytes im
                                      Datensegment
note:       .word 0                 # Integer mit Vorzeichen
                                      (mit Null initialisiert)
noten:      .word 256, 0x100        # 2 Integer ...


            .text                   # Es folgt der Programmcode

            .globl main             # main ist globales Symbol
main:       ...
```

Systemaufrufe

Die Nummer des Systemaufrufes (system call) wird im Register **$v0** übergeben:

```
li $v0, Nummer
syscall
```

Bildschirmausgabe

Dienst	Nummer	Eingabe
print_int	1	Integer in `$a0`
print_float	2	float in `$f12`
print_double	3	double in `$f12` und `$f13`
print_string	4	Adresse des String in `$a0`

Systemaufrufe

Tastatureingabe

Dienst	Nummer	(Ein-)Ausgabe
read_int	5	Integer in `$v0`
read_float	6	float in `$f0`
read_double	7	double in `$f0 und $f1`
read_string	8	Adresse der Zeichenkette in `$a0` und maximale Länge in `$a1` übergeben. Zeichenkette ist nullterminiert. Bei weniger eingegebenen Zeichen wird die Zeichenkette zusätzlich mit "\n" abgeschlossen

Sonstige Systemaufrufe:

Dienst	Nummer	Eingabe	Ausgabe
sbrk	9	Byte-Anzahl in `$a0`	Anfangsadresse in `$v0`
exit	10		

Beispiel

```
        .data
string: .asciiz "Hello MIPS-World"

        .text
        la $a0, string    # Adresse von string in $a0
        li $v0, 4         # print_string
        syscall
```

```
li $v0, Nummer
syscall
```

print_string	4	Adresse des String in `$a0`

Ausgabe einer Integerzahl mit LF

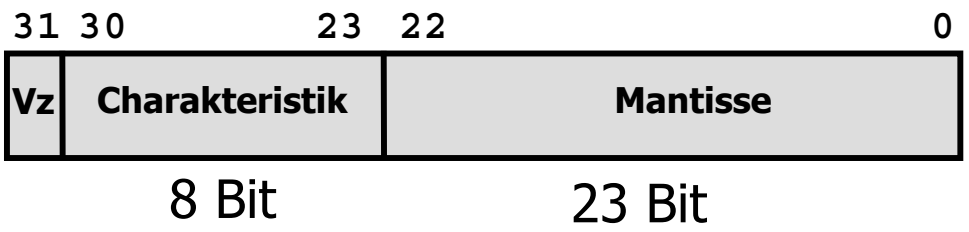
```
                .data
lf_string:    .asciiz "\n"

                .text
pr_str:         li $v0, 1           # print_int
                syscall
                la $a0, lf_string   # print_string
                li $v0, 4
                syscall
                jr $ra
```

Datenformate im MIPS-Prozessor

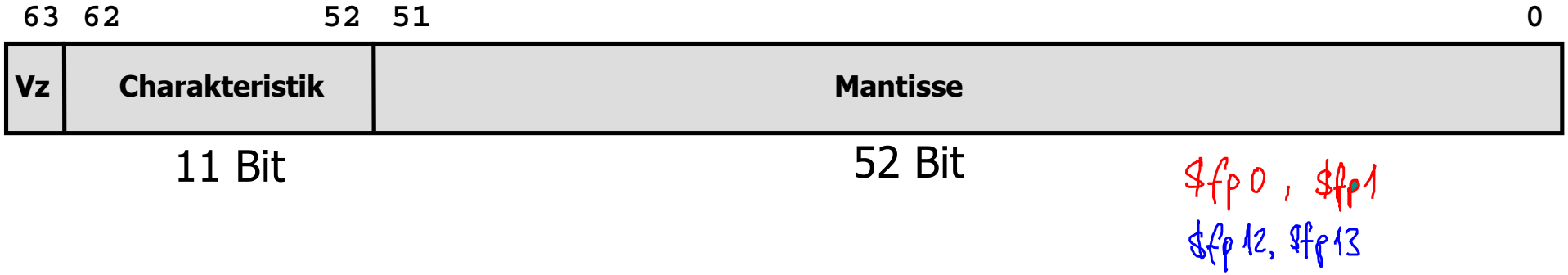
- Es sind folgende Datenformate definiert:
  - Byte (8 Bit), Halbwort (16 Bit), Wort (32 Bit)
- Ordnung von Bytes in Wörtern und Wörter in mehrfachen Wortstrukturen
  - Little Endian Order oder
  - Big Endian Order
- Vorzeichenbehaftete Zahlen werden in Zweierkomplement-Form dargestellt
- Ganze Zahlen werden entweder vorzeichenlos (unsigned) oder vorzeichenbehaftet (signed) in Zweierkomplement-Form dargestellt

Fließkommaformate

Einfache Genauigkeit:



Doppelte Genauigkeit:

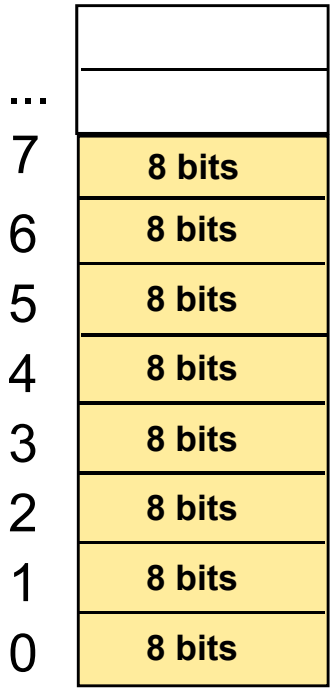


$fp0, $fp1
$fp12, $fp13

Bei Arithmetik mit doppelter Genauigkeit (64 Bit) dürfen nur Register mit gerader Registernummer verwendet werden!

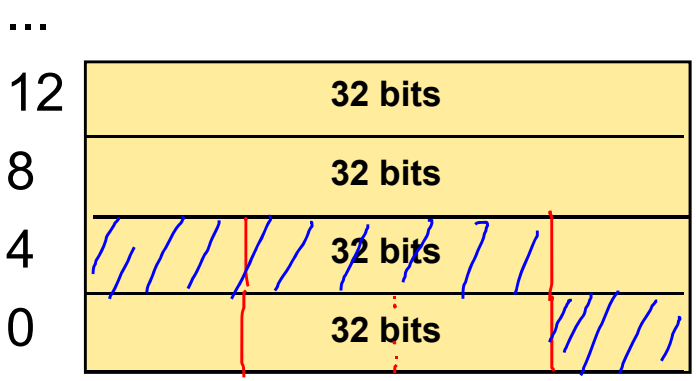
Speichermodell

- Speicher: Eindimensionales Array aus Speicherzellen mit Adressen.
- Speicheradresse: Index im Array
- "Byte-Adressierung" heißt, dass der Index auf ein Byte im Speicher zeigt.



Speichermodell

MIPS: Wort mit 32 Bit oder 4 Bytes

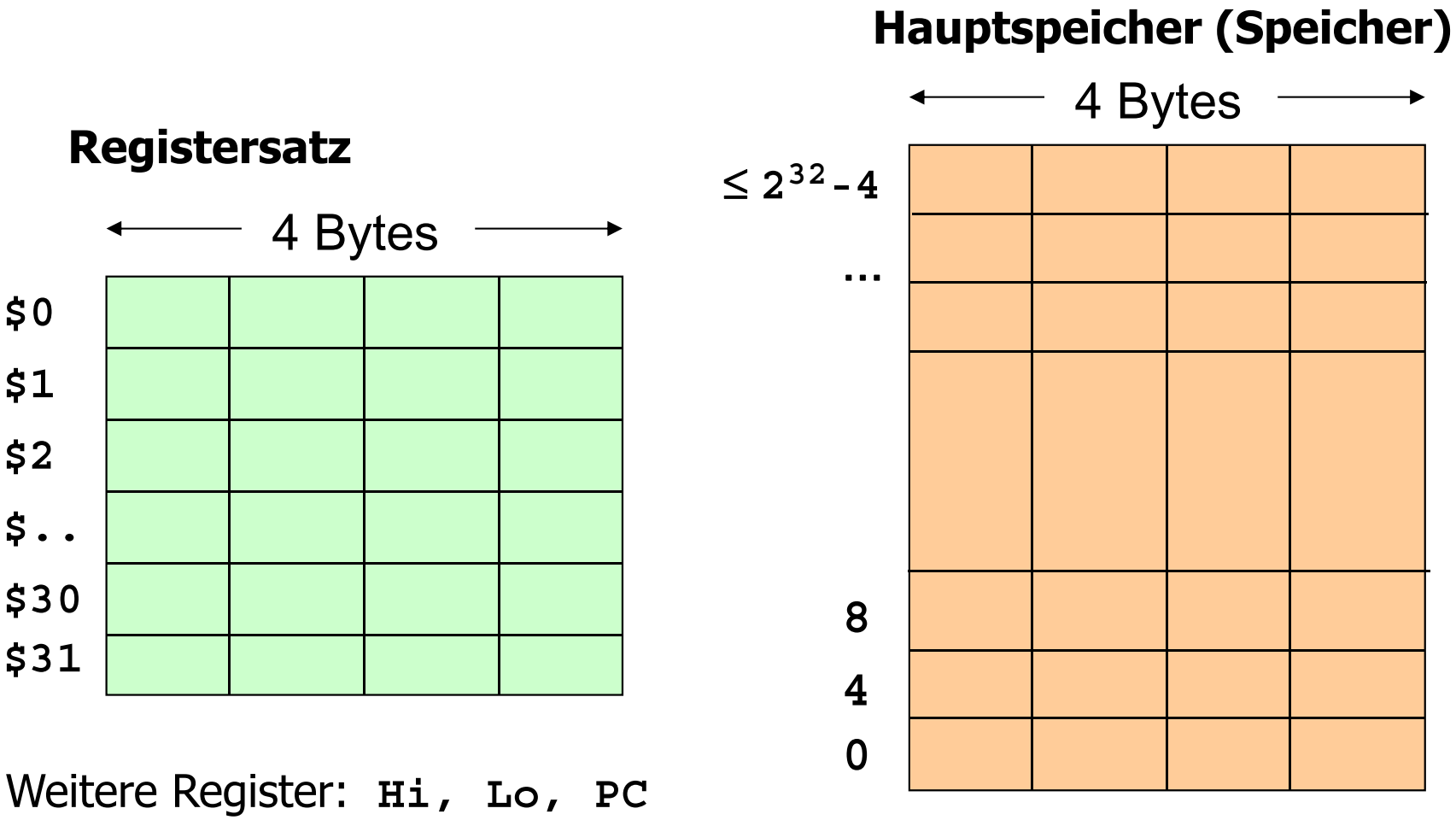


Es werden Wörter geladen, aber Byte adressiert.

- 2^{32} Byte mit Byte-Adressen von 0 bis $2^{32}-1$
- 2^{30} Wörter mit Byte-Adressen 0, 4, 8, ... $2^{32}-4$
- Wörter sind ausgerichtet.

Welche Werte haben die 2 niederwertigsten Bit einer Wort-Adresse?

Speichermodell



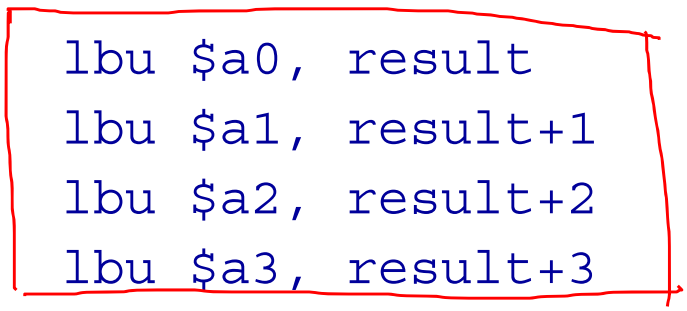
Weitere Register: `Hi, Lo, PC`

Speichermodell „Big Endian"

Big Endian:

Höchstwertigstes Byte liegt an kleinster Adresse

```
result:     .word 0x89abcdef
```




```
    lbu $a0, result
    lbu $a1, result+1
    lbu $a2, result+2
    lbu $a3, result+3
    # $a0 enthaelt 0x89
    # $a1 enthaelt 0xab
    # $a2 enthaelt 0xcd
    # $a3 enthaelt 0xef
```

MARS auf x86-System: Little Endian!

Speichermodell „Little Endian"

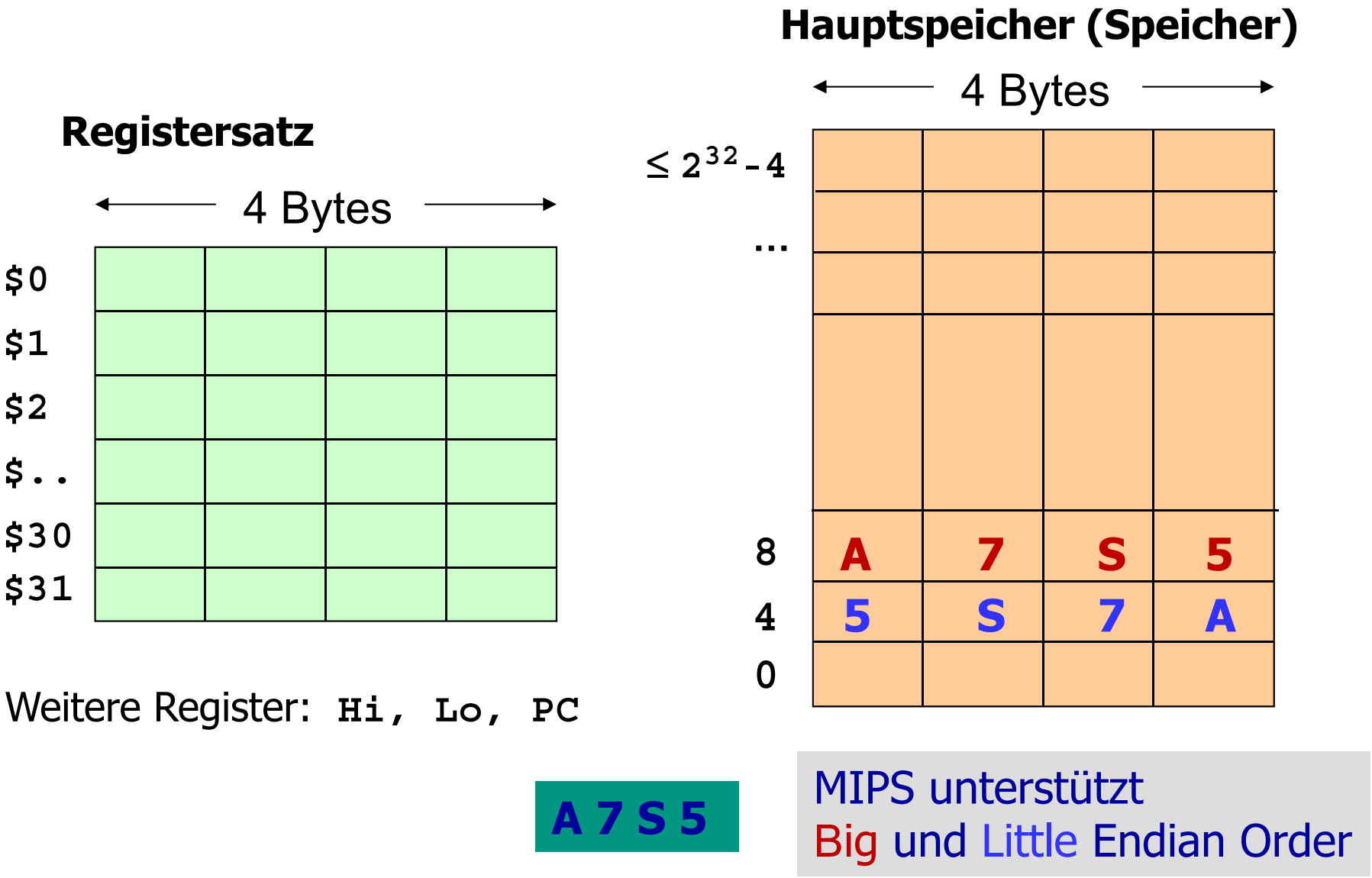
Little Endian:

Höchstwertigstes Byte liegt an größter Adresse

```
result:     .word 0x89abcdef
                  4. 3. 2. 1.

            lbu $a0, result
            lbu $a1, result+1
            lbu $a2, result+2
            lbu $a3, result+3
            # $a0 enthaelt 0xef
            # $a1 enthaelt 0xcd
            # $a2 enthaelt 0xab
            # $a3 enthaelt 0x89
```

MARS auf x86-System: Little Endian!

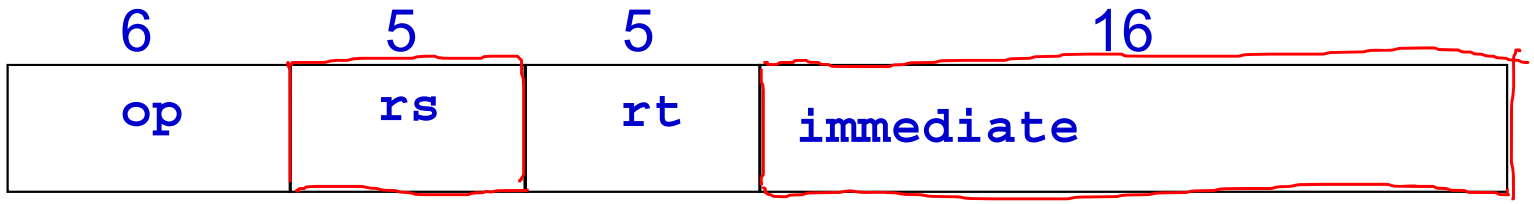
Speichermodel



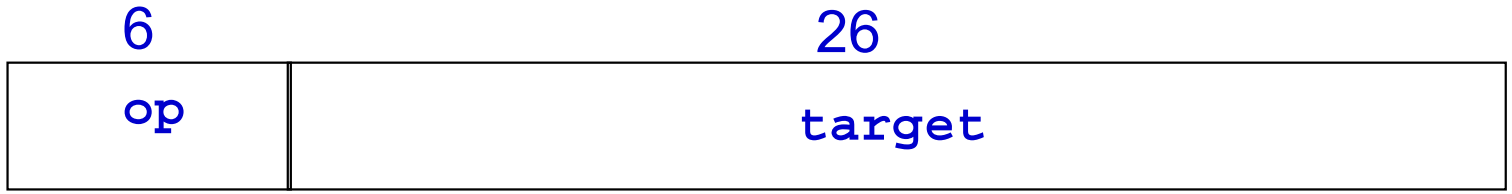
Befehlsformate

Der MIPS-Prozessor hat ausschließlich Befehle fester Länge (32-Bit). Die Befehle werden in Typ I, J und R unterteilt:

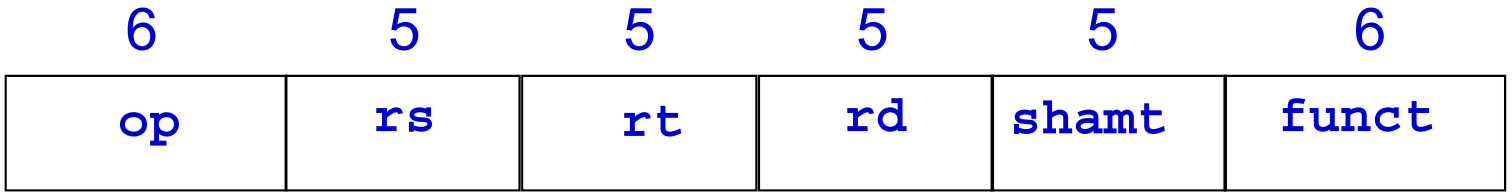
Typ I



Typ J



Typ R



Befehlsformate

Abk.	Bedeutung
I	Immediate (direkt)
J	Jump (Sprung)
R	Register
op	6 Bit OpCode des Befehls
rs	5 Bit Kodierung eines Quellenregisters
rt	5 Bit Kodierung eines Quellenregisters oder Zielregisters
immediate	16 Bit unmittelbarer Wert oder Adressverschiebung
target	26 Bit Sprungadresse
rd	5 Bit Kodierung des Zielregisters
shamt	5 Bit Kodierung der Größe einer Verschiebung
funct	6 Bit Kodierung der Funktion

Adressierungsarten des MIPS-Prozessors

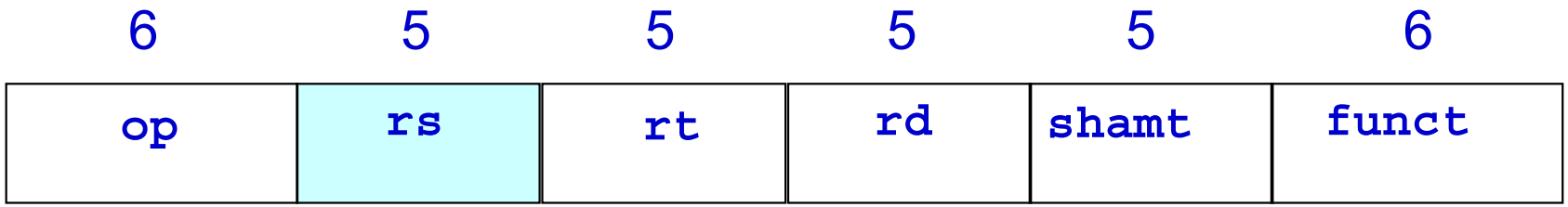
Karlsruher Institut für Technologie

Der MIPS-Prozessor unterstützt vier Adressierungsarten:

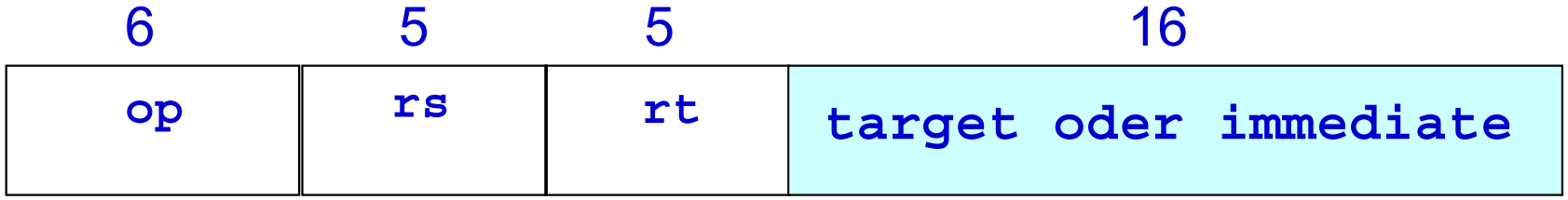
- **Explizite Register-Adressierung:** Der Operand steht in einem Register
- **Direkte Adressierung:** Der Operand ist eine Konstante im Befehlswort
- **Basis-Adressierung:** Der Operand ist im Speicher an der Adresse, die sich durch die Addition eines Registerinhalts und einer Konstanten im Befehl ergibt
- **Befehlszähler-relative Adressierung:** Die Adresse ergibt sich aus dem Wert des Befehlszählers und einer Konstanten im Befehl

Adressierungsarten des MIPS-Prozessors

Registeradressierung: (register)



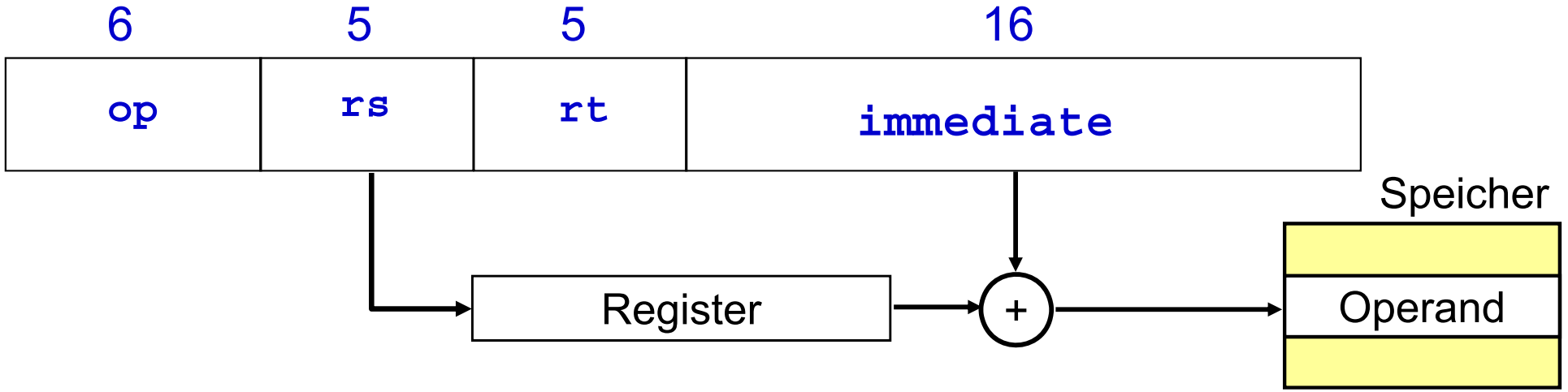
Direkte Adressierung: imm



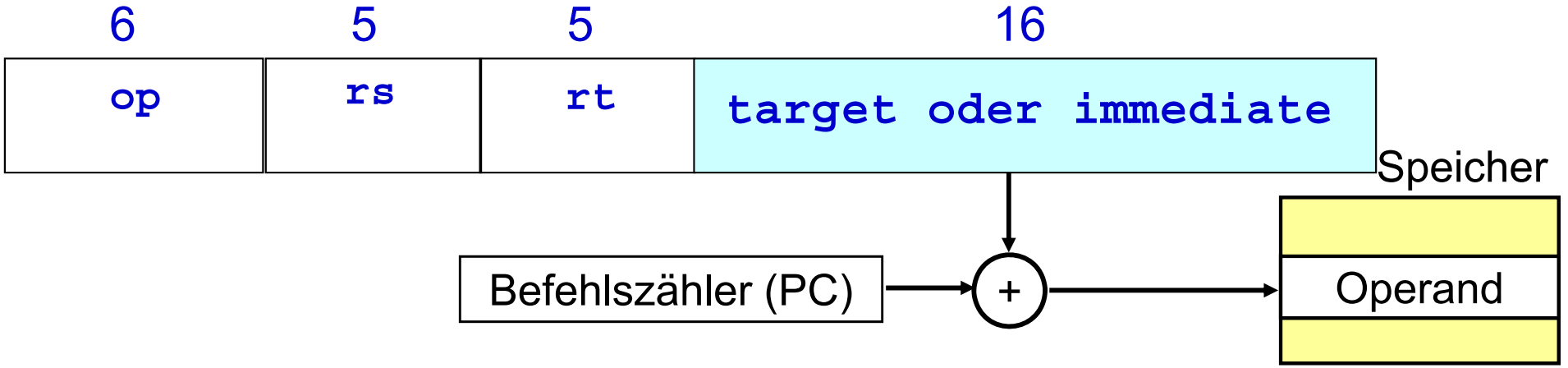
Adressierungsarten des MIPS-Prozessors

Basisadressierung: imm(register)

8($a0)



Befehlszähler-relative Adressierung: imm(PC)



Adressierungsarten des MIPS-Assembler

Format	Beispiel	Adressberechnung
(register)	`($s0)`	Inhalt des Registers
imm	`0x10003248` (hexadezimal)	unmittelbarer Wert
imm(register)	`0x23($s4)`	Inhalt des Registers + unmittelbarer Wert
symbol	`label1`	Adresse des Symbols
symbol ± imm	`marke+0x45`	Adresse des Symbols ± unmittelbarer Wert
symbol ± imm(register)	label + `0x13($s1)`	Adresse des Symbols ± (unmittelbarer Wert + Inhalt des Registers)

Befehlssatz

Arithmetische Befehle

abs $t0, $t1 := $t0 = |$t1|

- Absolutwert:
  `abs rdest,rsrc`
- Addition:
  `add rd,rs,rt` `addu rd,rs,rt` `addi rd,rs,imm`
- Division (Quotient in LO und Rest in HI)
  `div rs,rt` `divu rd,rs1,rs2`
- Multiplikation
  `mult rs,rt` `multu rs,rt` (unsigned)
  `mul rdest,rsrc1,rsrc2` `mulo rdest,rsrc1,rsrc2`
- Negation
  `neg rdest,rsrc` `negu rdest,rsrc`
- Subtraktion
  `sub rd,rs,rt` `subu rd,rs,rt`

Beispiel: Additionsbefehle in MIPS

Befehlsformate (z.B. bei der Addition):

Zielregister

```
add rd,rs,rt
```

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

```
addu rd,rs,rt
```

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

```
addi rt,rs,imm
```

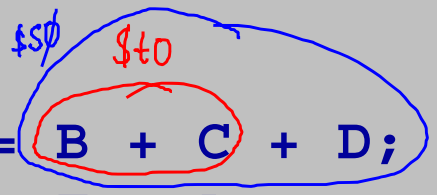
8	rs	rt	imm
6	5	5	16

Beispiel: Arithmetische Befehle

- Alle Befehle haben 3 Operanden
- Operand-Reihenfolge ist fest (Zielregister zuerst)
- Operanden einer arithmetischen Operation **müssen** in Registern stehen.
  Alle Register sind 32 Bit breit.

Beispiele:

C-Code:
```
A = B + C
```
MIPS-Code:
```
add $s0, $s1, $s2
```



C-Code:
```
A = B + C + D;
E = F - A;
```
MIPS-Code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```