

# MIPS-Assembler

---

## Assemblerprogrammierung mit dem MIPS-Simulator SPIM

- Der SPIM-Simulator (MIPS R2000/R3000-Prozessor)  
**Literatur: Hennessy & Patterson (Anhang A auf der TI-Homepage)**
- Programmiermodell des SPIM-Simulators
  - Aufbau eines MIPS-Prozessors
  - Registersatz
  - Speicheraufteilung
- MIPS-Assemblerprogrammierung
  - Syntax der MIPS-Assemblersprache
  - Assemblerdirektiven
  - Adressierungsarten
  - Datenformate
  - Befehlsformate & Befehlssatz



# Installation und Benutzung

---

- ❑ Simulator abrufbar unter:

**`http://i61www.ira.uka.de/users/asfour/TI/TI-2/Spim`**

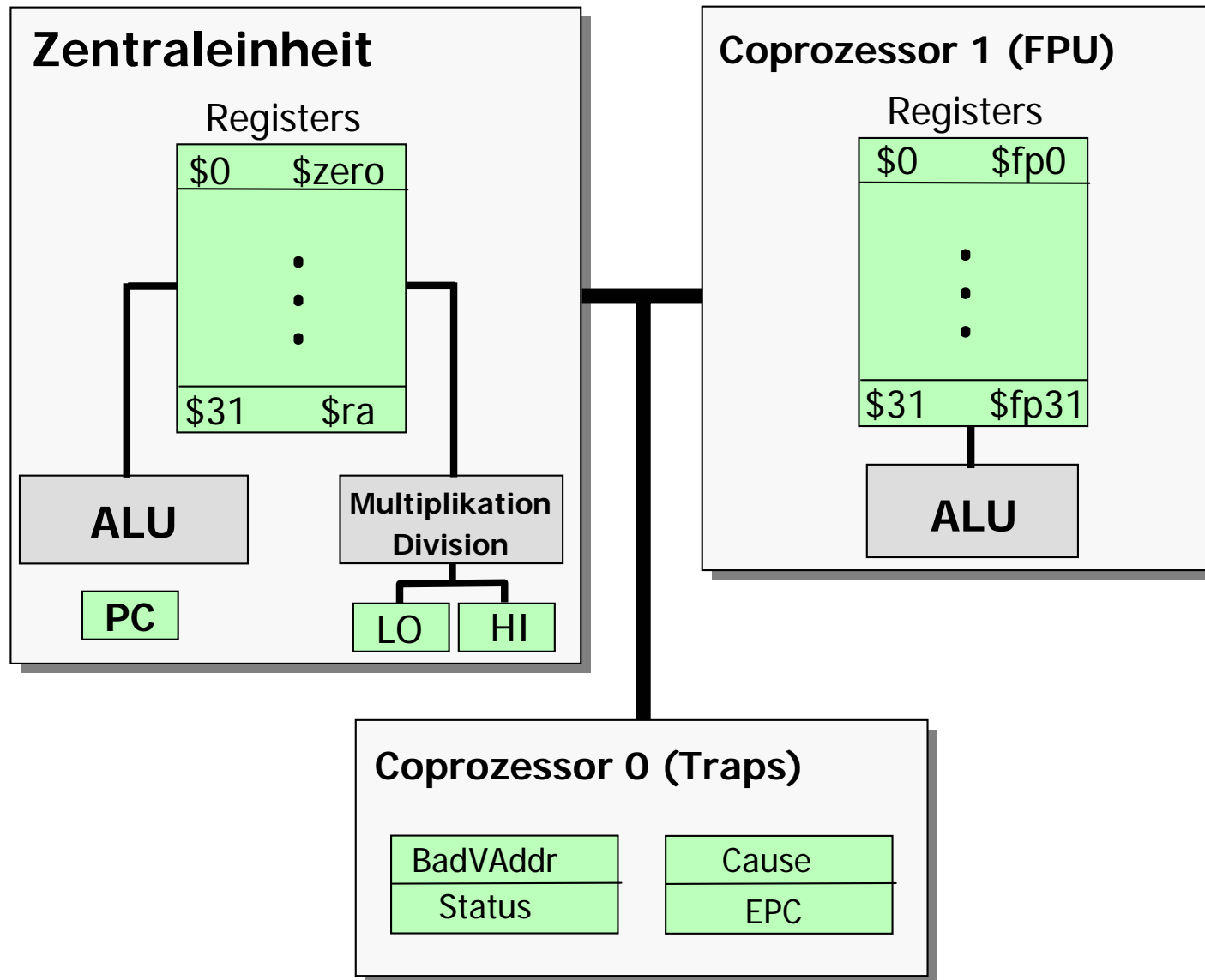
- ❑ SPIM unterliegt nur der GNU-Lizenz. Die neueste Version ist erhältlich unter:

**`ftp://ftp.cs.wisc.edu/pub/spim`**

- Unix-Versionen: `spim.tar.gz`, `spim.tar.z`
- Macintosh-Version: `SPIM.sit.bin` und `SPIM.Hqx.txt`
- Windows-Version: `spim.zip`



# Aufbau des MIPS-Prozessors



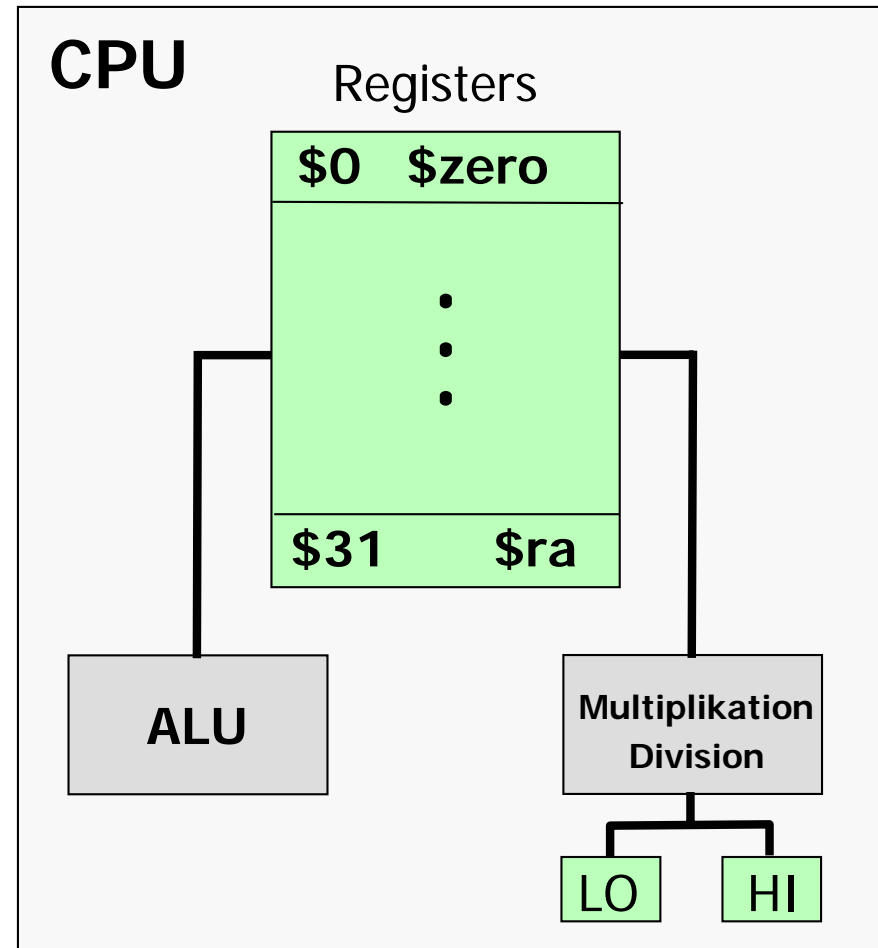
# Registersatz

## 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 verwendbare Register.

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 <b>nicht</b> 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. <b>vor Unterprogrammaufruf</b> gesichert werden
\$s0 – \$s7	\$16 - \$23	Register für langlebige Variablen. Sie müssen <b>vom Unterprogramm</b> gesichert werden

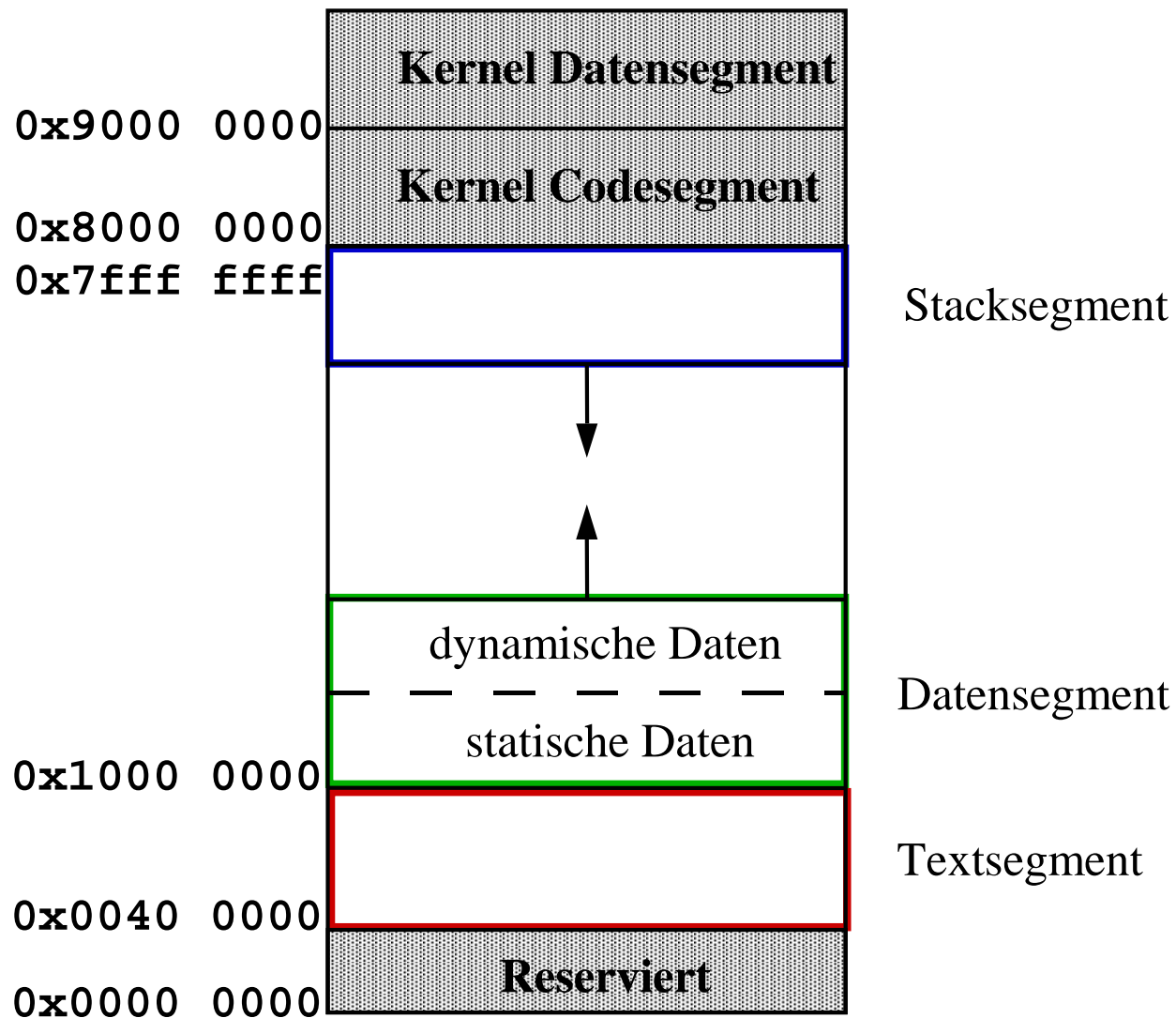


# Registersatz

Name	Nr.	Verwendung
\$k0 – \$k1	\$26 - \$27	Reserviert für das Betriebssystem. Dürfen vom Programmierer <b>nicht</b> 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	\$29	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

Grundsätzlich gilt:

**Jede Befehlszeile sollte kommentiert werden**



# MIPS-Assemblerdirektiven

---

Assemblerdirektiven beginnen mit einem Punkt (.)

## **.align n**

Die folgenden Daten sollen an der nächstmöglichen Adresse gespeichert werden, die durch **2<sup>n</sup>** teilbar ist.

## **.space n**

Allokiert **n** Bytes im aktuellen Segment, welches bei SPIM das Datensegment sein muss.

## **.ascii string**

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**  $b_1, \dots, b_n$

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

**.half**  $h_1, \dots, h_n$

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

**.word**  $w_1, \dots, w_n$

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

**.float**  $f_1, \dots, f_n$

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



# MIPS-Assemblerdirektiven

---

**.double  $d_1, \dots, d_n$**

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

**.globl symbol**

Deklariert das Datum, welches bei der Marke **symbol** gespeichert ist, als globales Symbol.

**.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>`

Folgende Daten sollen im (Kernel-)Datensegment gespeichert werden. Optional kann eine Adresse `<addr>` angegeben werden. Der Assembler beginnt ab Adresse `1001 000016` Daten im Datensegment abzulegen.

## `.(k)text <addr>`

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 16,0x100               # 2 Integers ...

.text                                    # Es folgt der Programmcode

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

# Eingabe und Ausgabe: Systemaufrufe

Die Nummer des Systemaufrufes (system calls) wird ins 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 Zeichenkette zusätzlich mit "\n" abgeschlossen



# Systemaufrufe

## Sonstiges 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 CR

---

```
                .data

cr_string:      .ascii "\n"

                .text

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

# Datenformate im MIPS-Prozessor

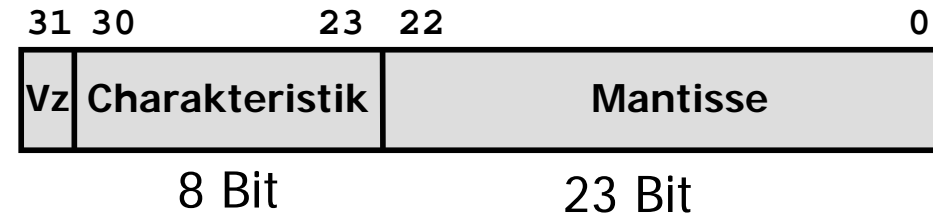
---

- ❑ 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
- ❑ Vorzeichenbehaftet Zahlen werden in Zweierkomplement-Form dargestellt
- ❑ Ganze Zahlen werden entweder vorzeichenlos (unsigned) oder vorzeichenbehaftet (signed) in Zweierkomplement-Form dargestellt

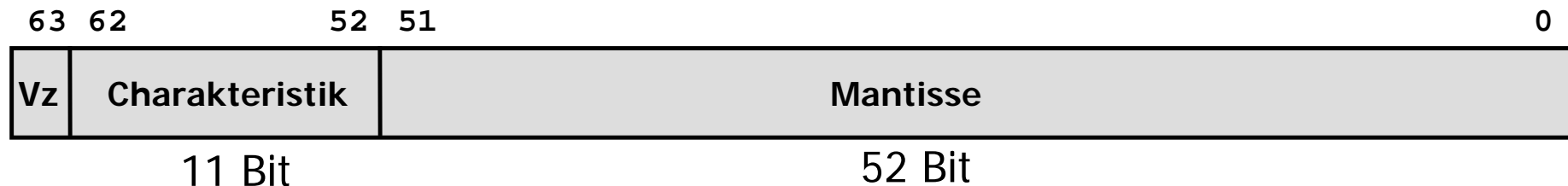


# Fließkommaformate

Einfache Genauigkeit:



Doppelte Genauigkeit:



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



# Das Speichermodell

MIPS: Wort mit 32 Bit oder 4 Bytes

...

12	32 bits
8	32 bits
4	32 bits
0	32 bits

Es werden Wörter  
geladen, aber Bytes  
adressiert.

- $2^{32}$  Bytes 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 niedrigstwertigen Bits einer Wort-Adresse?**



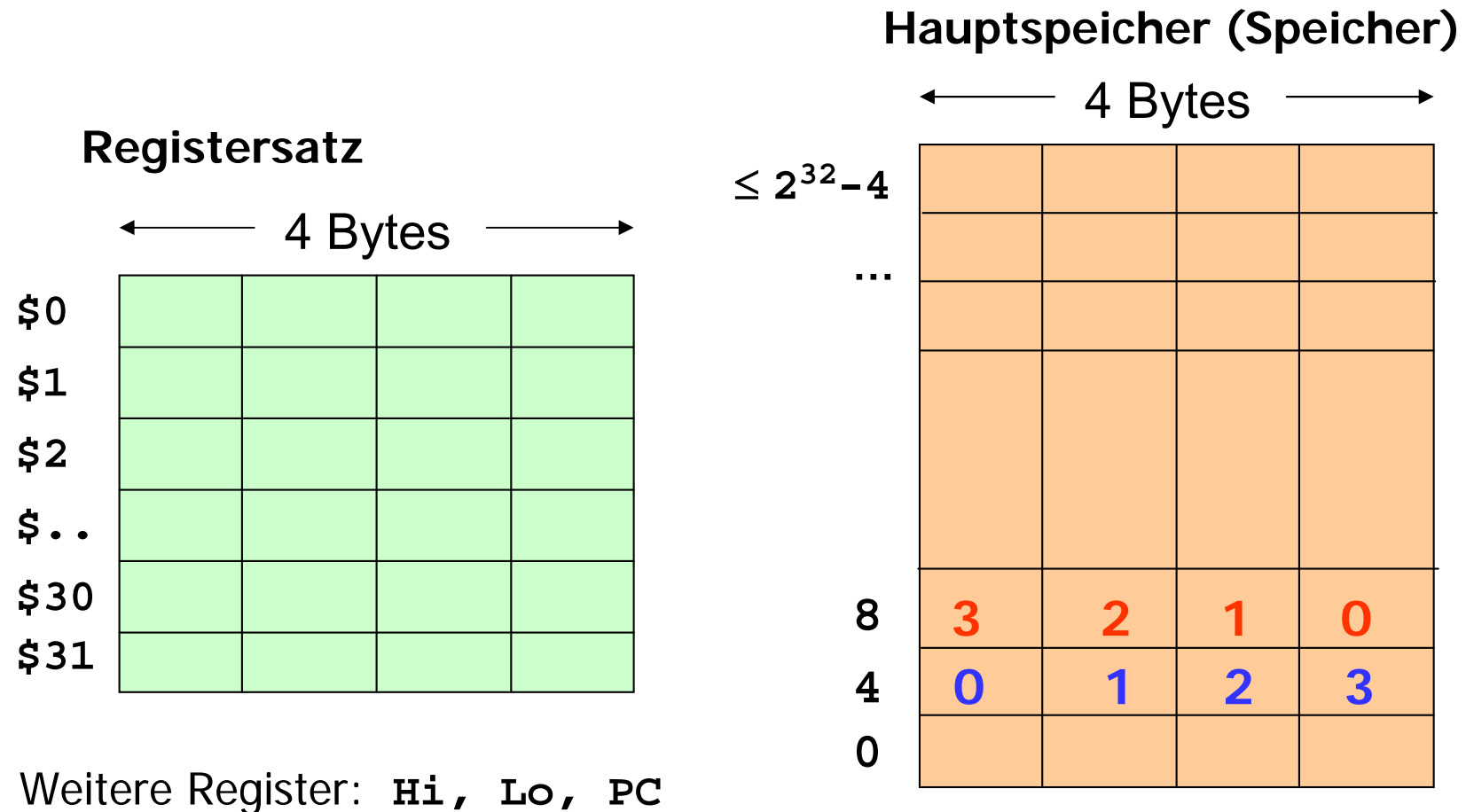
# Das Speichermodell

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

...	
7	8 bits
6	8 bits
5	8 bits
4	8 bits
3	8 bits
2	8 bits
1	8 bits
0	8 bits



# Das Speichermodell der MIPS-Architektur

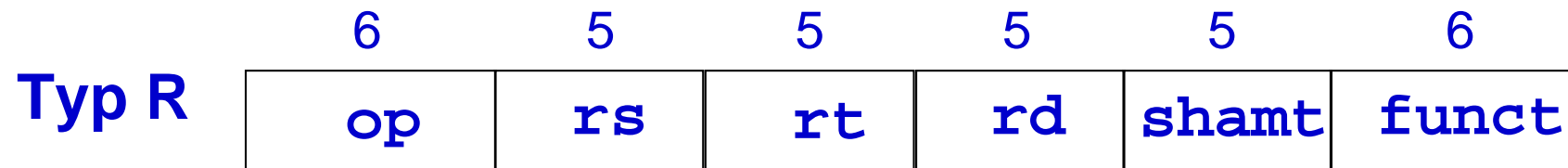
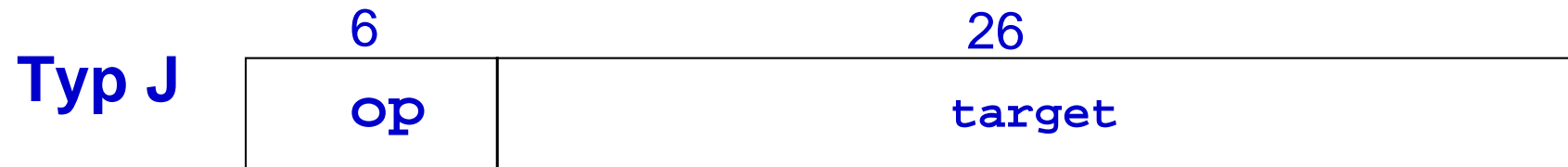
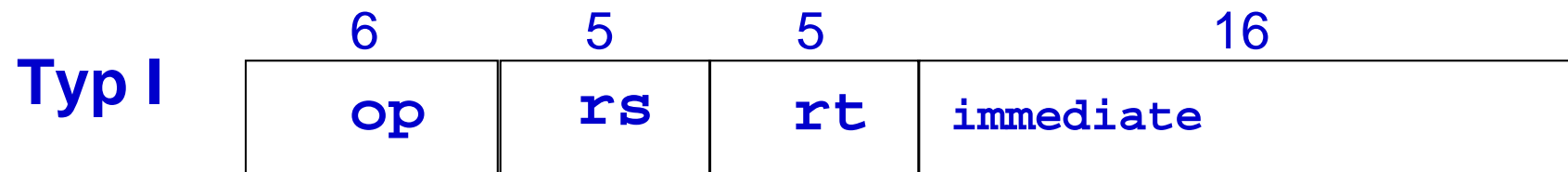


MIPS unterstützt  
**big** und **little** endian order



# Befehlsformate

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



# 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

---

Der MIPS-Prozessor unterstützt vier Adressierungsarten:

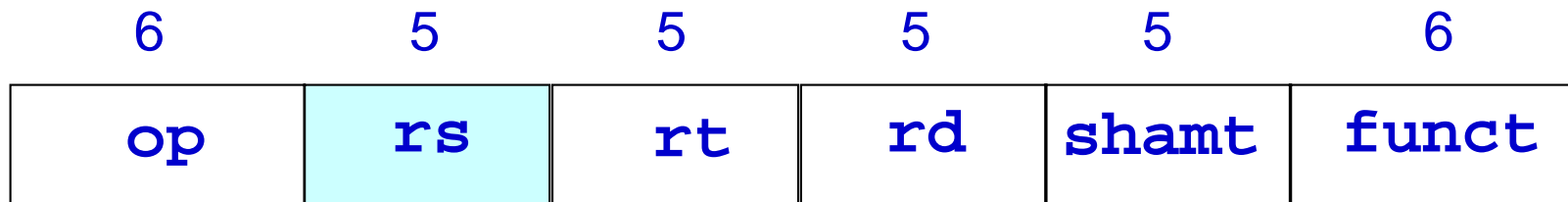
- ❑ **Explizite Register-Adressierung:** Der Operand steht in einem Register
- ❑ **Direkte Adressierung:** Der Operand ist eine Konstante im Befehlswort
- ❑ **Basis-Adressierung:** Der Operand ist im Speicher an der Adresse, die sich durch die Addition eines Registerinhalts und einer Konstanten im Befehl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



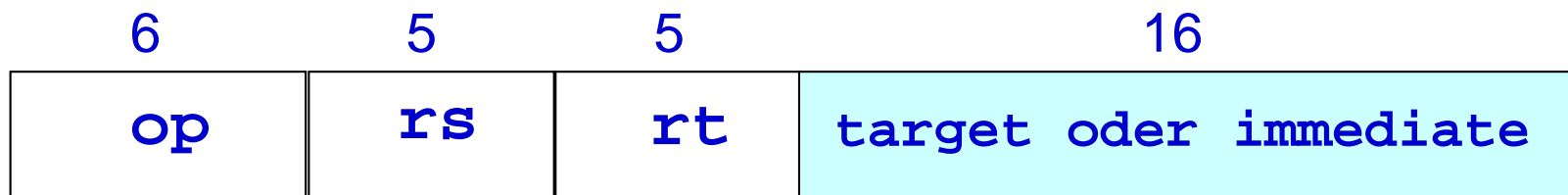
# Adressierungsarten des MIPS-Prozessors

---

Registeradressierung: (register)

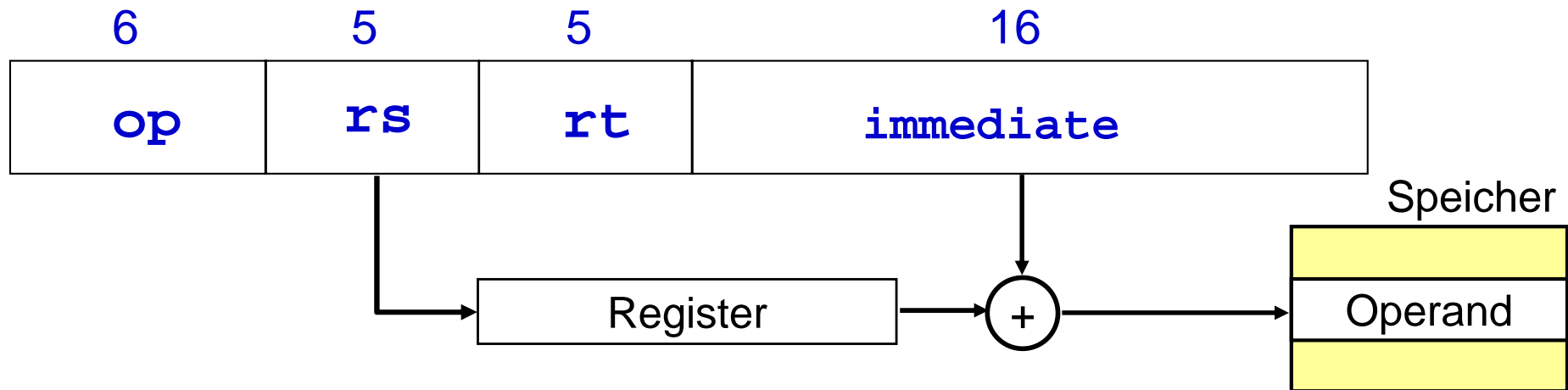


Direkte Adressierung: imm

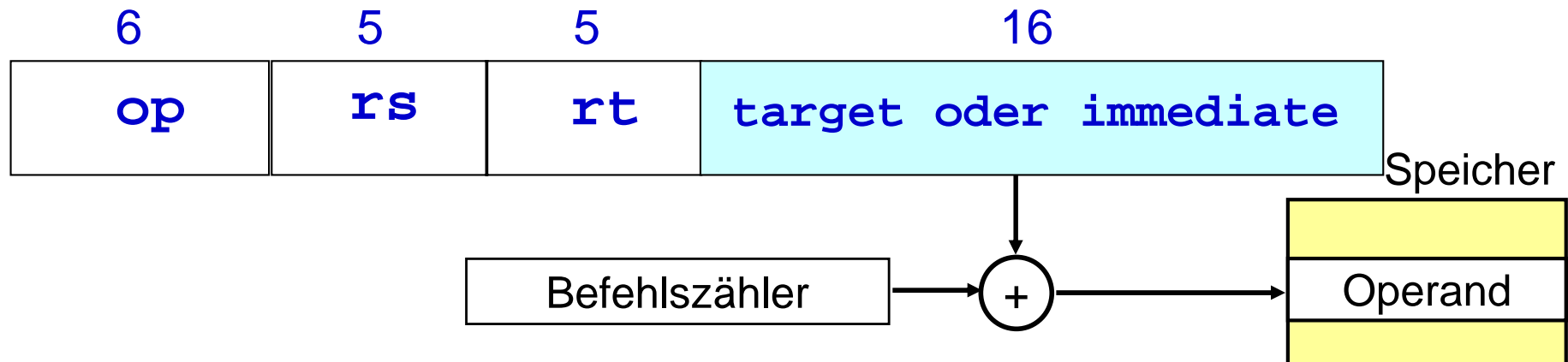


# Adressierungsarten des MIPS-Prozessors

Basisadressierung:  $\text{imm}(\text{register})$



Befehlszähler-relative Adressierung:  $\text{imm}(\text{PC})$



# Adressierungsarten in SPIM

Format	Beispiel	Adressberechnung
(register)	( <b>\$s0</b> )	Inhalt des Registers
imm	<b>0x10003248</b>	unmittelbarer Wert
imm(register)	<b>0x23(\$s4)</b>	Inhalt des Registers + unmittelbarer Wert
symbol	<b>label1</b>	Adresse des Symbols
symbol $\pm$ imm	<b>marke+0x45</b>	Adresse des Symbols $\pm$ unmittelbarer Wert
symbol $\pm$ imm(register)	label + <b>0x13(\$s1)</b>	Adresse des Symbols $\pm$ (unmittelbarer Wert + Inhalt des Registers)



# Befehlssatz

## Arithmetische Befehle

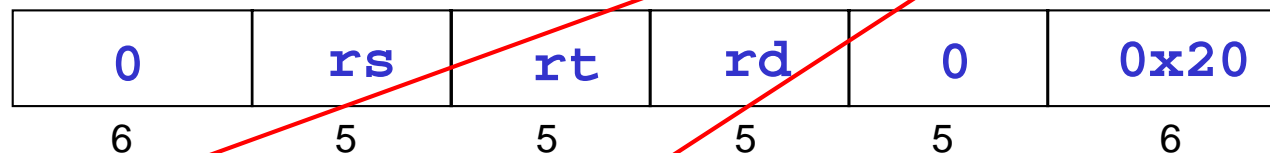
- Absolutwert:  
`abs rdest, rsrc`
- Addition:  
`add rd,rs,rt`      `addu rd,rs,rt`      `addi rd,rs,imm`
- Division (Quotient in LO und Rest in HI)  
`div rs,rt`      `divu rd,rs1,rs2`
- Multiplikation  
`mult rs, rt`      `multu rs, rt (unsigned)`  
`mul rdest,rsrc1,rsrc2`      `mulo rdest,rsrc1,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

**add rd,rs,rt**



**addu rd,rs,rt**



**addi rt,rs,imm**



# 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:	<code>add \$s0, \$s1, \$s2</code>

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



# Struktur eines MIPS-Programms

---

```
.data
# globale Daten
.text
# Unterprogramme

.globl main
main:    subu $sp, $sp, 8      # Stack Frame ist 8 Bytes
        sw $ra, 4($sp)       # Sichern der Ruecksprungadresse
        sw $fp, 8($sp)       # Sichern des alten Frame-Pointers
        addu $fp, $sp, 8     # neuen Frame-Pointer definieren

# Hauptprogramm

        lw $ra, 4($sp)       # Ruecksprungadresse wiederherstellen
        lw $fp, 8($sp)       # Frame-Pointer wiederherstellen
        addu $sp, $sp, 8     # Stack-Frame loeschen
        jr $ra
```



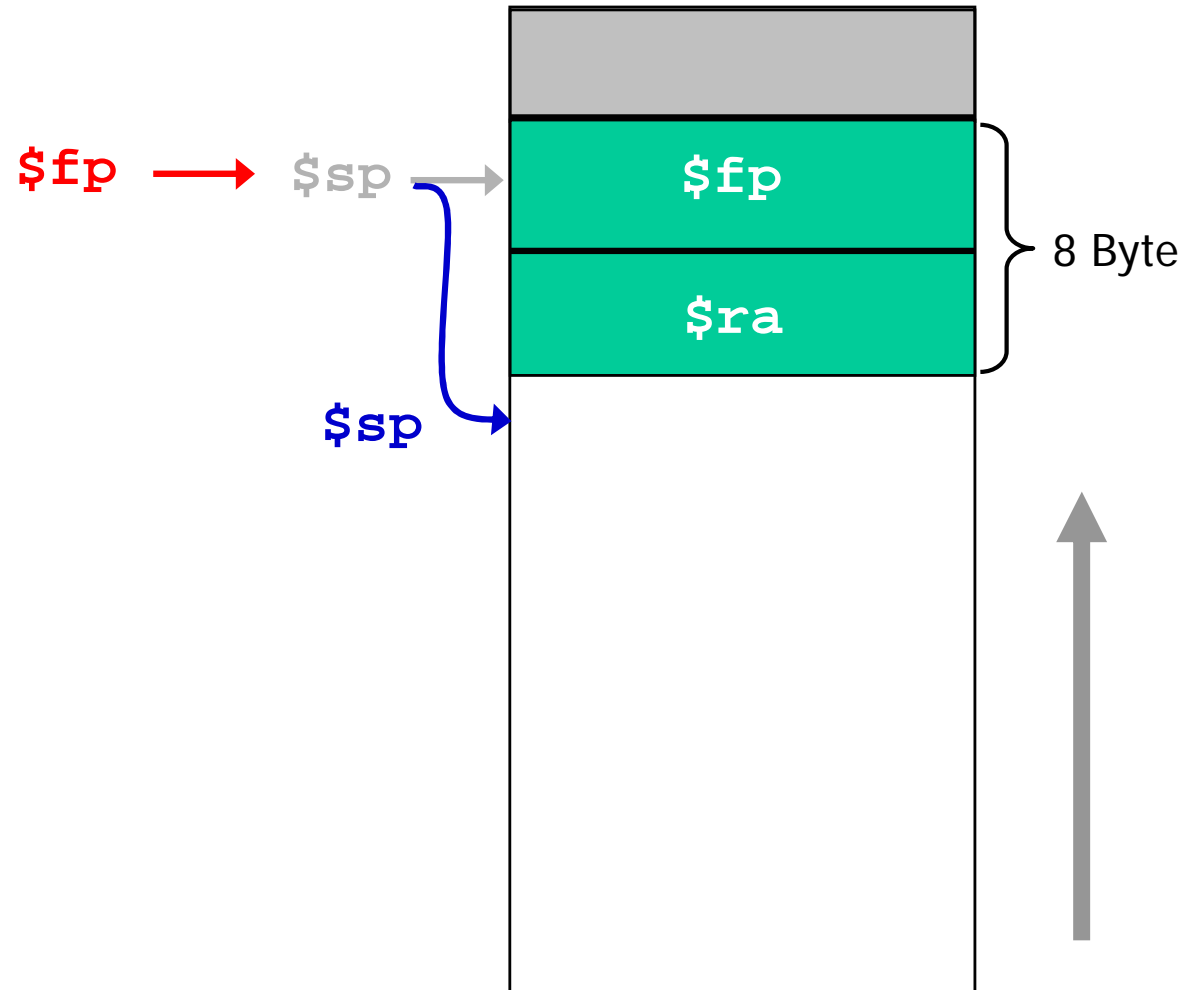
# Struktur eines MIPS-Programms

```
.data
# globale Daten
.text
# Unterprogramme

.globl main
main: subu $sp, $sp, 8
      sw $ra, 4($sp)
      sw $fp, 8($sp)
      addu $fp, $sp, 8

      # Hauptprogramm

      lw $ra, 4($sp)
      lw $fp, 8($sp)
      addu $sp, $sp, 8
      jr $ra
```



# Beispiel: Integer-Arithmetik

```
.data
cr_string:    .ascii "\n"           # Sonderzeichen "neue Zeile"
eingabeA:     .ascii "Integer-Zahl A: "
eingabeB:     .ascii "Integer-Zahl B: "
result_sum:   .ascii "A + B = "
result_dif:   .ascii "A - B = "
result_mul:   .ascii "A * B = "
result_div:   .ascii "A mod B = "
result_rst:   .ascii "Rest = "
error_str:    .ascii "Division durch Null nicht definiert!\n"
```

## .text

```
# Prozedur: Ausgabe eine Integer-Zahl mit CR
print_int:    li $v0, 1
              syscall
              la $a0, cr_string
              li $v0, 4
              syscall
              jr $ra
```



# Beispiel: Integer-Arithmetik

```
# Prozedur: Ausgabe eines Strings
print_str:    li $v0, 4
              syscall
              jr $ra

              .globl main
main:         subu $sp, $sp, 8      # Stack Frame ist 8 Bytes
              sw $ra, 4($sp)       # Sichern der Ruecksprungadresse
              sw $fp, 8($sp)       # Sichern des alten Frame-Pointers
              addu $fp, $sp, 8     # neuen Frame-Pointer definieren

              la $a0, eingabeA     # Integer-Zahl A holen
              jal print_str
              li $v0, 5
              syscall
              move $s0, $v0        # A in $s0 sichern

              la $a0, eingabeB     # Integer-Zahl B holen
              jal print_str
              li $v0, 5
              syscall
              move $s1, $v0        # B in $s1 sichern
```



# Beispiel: Integer-Arithmetik

```
la $a0, result_sum      # Ausgabe A + B
jal print_str
add $a0, $s0, $s1
jal print_int

la $a0, result_dif      # Ausgabe A - B
jal print_str
sub $a0, $s0, $s1
jal print_int

la $a0, result_mul      # Ausgabe A * B
jal print_str
mul $a0, $s0, $s1
jal print_int

beqz $s1, error
la $a0, result_div      # Ausgabe A mod B
jal print_str
div $s0, $s1
mflo $a0
jal print_int
```



# Beispiel: Integer-Arithmetik

```
la $a0, result_rst # Ausgabe des Restes
jal print_str
mfhi $a0
jal print_int
b fertig
```

```
error:      la $a0, error_str # Division durch Null
            jal print_str
```

```
fertig:    lw $ra, 4($sp)      # Ruecksprungadresse wiederherstellen
            lw $fp, 8($sp)    # Frame-Pointer wiederherstellen
            addu $sp, $sp, 8   # Stack-Frame loeschen
            jr $ra
```

