

# Übung 2

---

## □ Einführung: Programmieraufgabe

## □ MIPS-Assembler

- Der SPIM-Simulator (MIPS R2000/R3000-Prozessor)
- Programmiermodell des SPIM-Simulators
- MIPS-Assemblerprogrammierung
- ...



# Programmieraufgabe

---

## □ Programm-Darstellung

- Symbolische Darstellung
- Maschinencode-Darstellung

## □ Programm-Übersetzung

- Assemblersprache
- Assembleranweisungen
- Assemblierung



# Programmieraufgabe

---

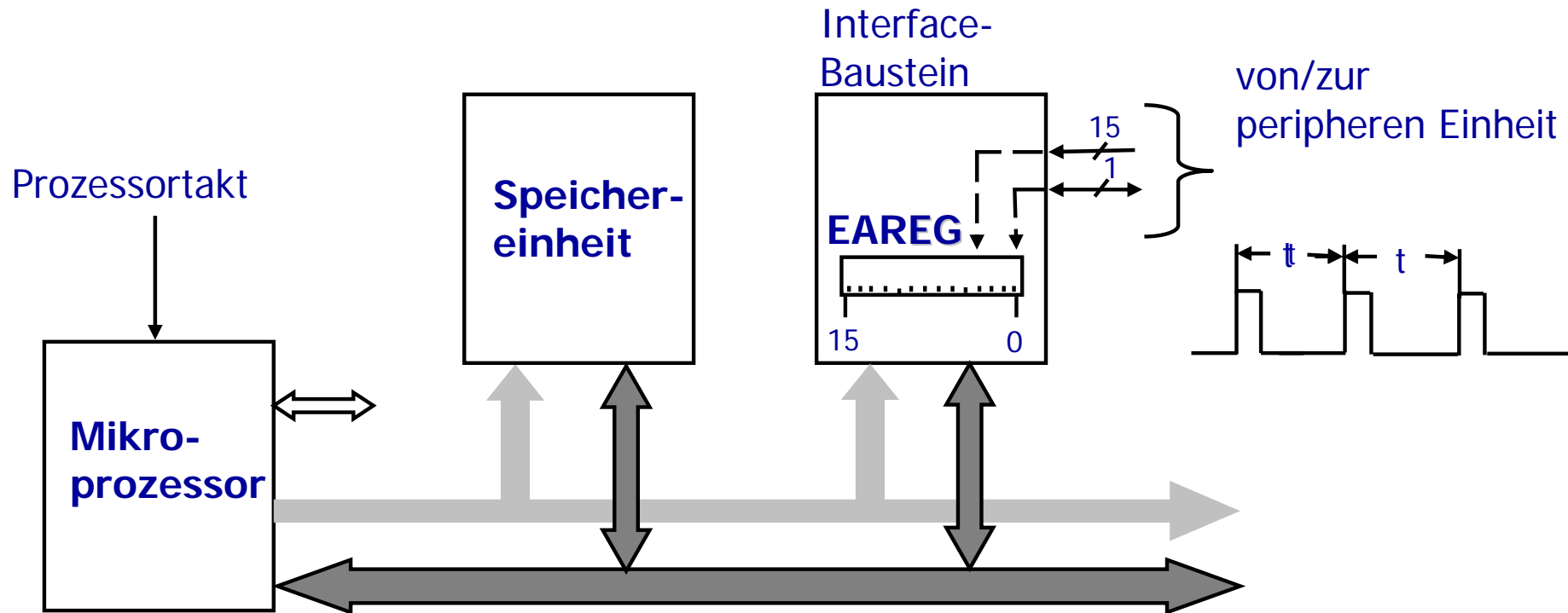
## Aufgabenstellung

Es soll ein Impulsgeber mit Hilfe eines  $\mu$ P-Systems aufgebaut werden, der in konstanten Zeitabständen Impulse an eine Ein-/Ausgabeeinheit abgibt

- Festlegen eines Satzes von Maschinenbefehlen zur Lösung dieser Aufgabe
- Erstellung des Programms in der Assemblersprache und in der Maschinensprache



# Programmieraufgabe

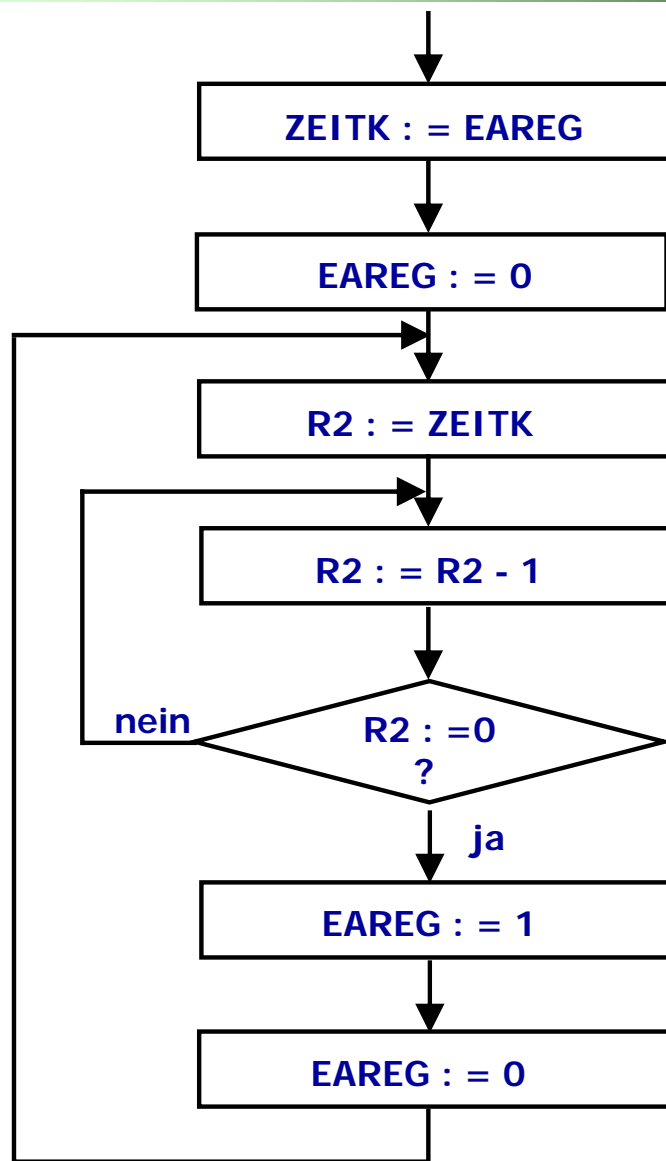


EA-Einheit mit einem 16 Bit Register **EAREG**

Absolute Adresse von EAREG: **\$8000 (32768)**

Periodendauer der Impulsfolge:  **$t$**

# Programmablauf in Form eines Flussdiagramms



- Periodendauer aus EAREG in die Hauptspeicherzelle **ZEITK**
- **EAREG** mit **NULL** initialisieren  
(**NULL** und **EINS** sind Speicherzellen mit konstanten Operanden)
- Periodendauer ist bestimmt durch die Anzahl der Durchläufe der innere Schleife und die Verarbeitungszeiten der einzelnen Befehlen

# Notwendige Befehle zur Lösung

## **MOVE QADR, ZADR**

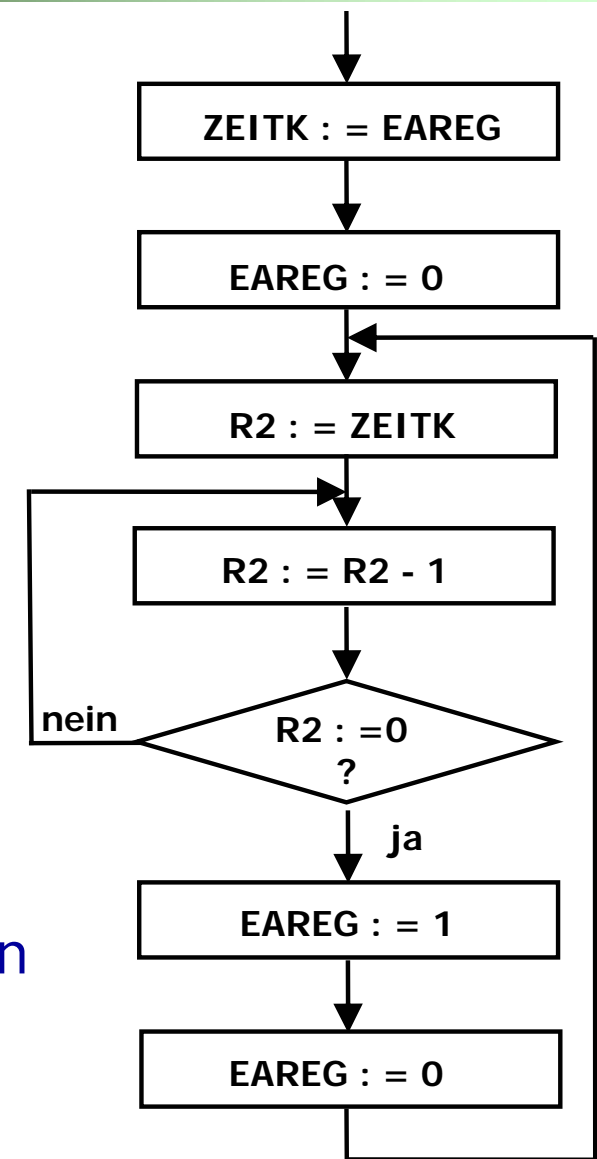
Inhalt von QADR (Quelladresse)  
nach ZADR (Zieladresse)

## **SUB QADR, ZADR**

Subtrahiere den Inhalt von QADR  
vom Inhalt von ZADR und schreibe  
das Ergebnis in ZADR

## **CMP ADR1, ADR2**

Vergleiche die mit ADR1 und ADR2 adressierten  
Operanden. Ergebnis in den CC-Bits des  
Prozessor-Statusregisters



# Notwendige Befehle zur Lösung

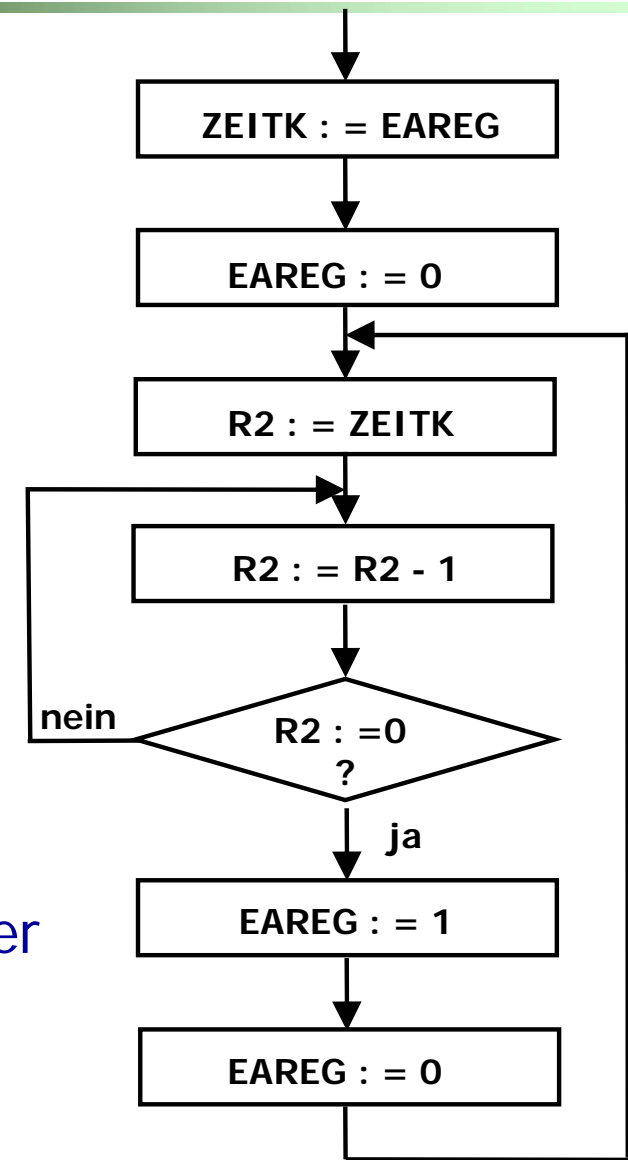
## BNE SPRADR

### Bedingter Sprung:

lade den Befehlszähler mit der Sprungadresse SPRADR, sofern das Prozessor-Statusregister den Zustand "ungleich" zeigt, sonst wird der nächste Befehl im Programm ausgeführt

## JMP SPRADR

**Unbedingter Sprung:** lade den Befehlszähler mit der Sprungadresse SPRADR



# Programm in symbolischer Darstellung

	MOVE	EAREG, ZEITK	6
	MOVE	NULL, EAREG	6
	MOVE	NULL, R0	4
MARKE1	MOVE	ZEITK, R2	4
MARKE2	SUB	EINS, R2	5
	CMP	R0, R2	4
	BNE	MARKE2	3
	MOVE	EINS, EAREG	6
	MOVE	NULL, EAREG	6
	JMP	MARKE1	3
ZEITK			
NULL	0		
EINS	1		

↑  
Takte/Befehl

# Berechnung von ZEITK

---

Impulsabstand: 0,005 s

Taktzeit: 0,04  $\mu$ s (25 MHz)



# Maschinencode-Darstellung

**Symbolische Angaben durch ihre äquivalente binäre Darstellungen ersetzen:**

- Zuordnung der symbolischen Operationscodes zu binären Operationscodes liegt durch eine **Zuordnungstabelle** fest:

<b>Symbol</b>	<b>Binärcode</b>	
<b>MOVE</b>	<b>0000</b>	<b>0001</b>
<b>SUB</b>	<b>0000</b>	<b>0010</b>
<b>CMP</b>	<b>0010</b>	<b>0001</b>
<b>BNE</b>	<b>0000</b>	<b>1100</b>
<b>JMP</b>	<b>0000</b>	<b>0101</b>

# Maschinencode-Darstellung

---

- Die Ersetzung symbolischer Adressen durch numerische Adressen ergibt sich aus der Lage des Programms im Hauptspeicher
- Numerische Adressen von Registern liegen fest.
- Adresse von EAREG ist durch die Adressdecodierung der EA-Einheit vorgegeben (\$8000)

## Voraussetzung:

Programm belegt den Hauptspeicher ab der Zelle 0

➔ Adresszuordnung als Symboltabelle

# Maschinencode-Darstellung

```

MOVE EAREG, ZEITK
MOVE NULL, EAREG
MOVE NULL, R0
MOVE ZEITK, R2
MARKE1 SUB EINS, R2
MARKE2 CMP R0, R2
      BNE MARKE2
MOVE EINS, EAREG
MOVE NULL, EAREG
JMP  MARKE1

ZEITK
NULL  0
EINS  1
    
```

## Symboltabelle für unser Programm:

Symbol	Adresse						
	Dual				Dezimal	Hex.	
MARKE1	0000	0000	0000	1000	8	0008	
MARKE2	0000	0000	0001	1010	10	000A	
ZEITK	0000	0000	0001	0111	23	0017	
NULL	0000	0000	0001	1000	24	0018	
EINS	0000	0000	0001	1001	25	0019	
EAREG	1000	0000	0000	0000	32768	8000	

# Maschinencode-Darstellung

Adresse		Maschinencode		Symbol
Dez.	Dual	Dual	Hex.	
0	00000000	<u>00000001</u> 00000000	0100	MOVE
1	00000001	1000000000000000	8000	EAREG
2	00000010	0000000000010111	0017	ZEITK
3	00000011	<u>00000001</u> 00000000	0100	MOVE
4	00000100	0000000000011000	0018	NULL
5	00000101	1000000000000000	8000	EAREG
6	00000110	<u>00000001</u> 00001000	0108	MOVE
7	00000111	0000000000011000	0018	NULL
8	00001000	<u>00000001</u> 00001010	010A	MOVE
9	00001001	0000000000010111	0017	ZEITK
10	00001010	<u>00000001</u> 000001010	020A	SUB
11	00001011	0000000000011001	0019	EINS
12	00001100	0010000110001010	218A	CMP
13	00001101	0000110000000000	0C00	BNE
14	00001110	00000000000 <u>1010</u>	000A	M2
15	00001111	0000000100000000	0100	MOVE
16	00010000	0000000000011001	0019	EINS
17	00010001	<u>10000000</u> 00000000	8000	EAREG
18	00010010	0000000100000000	0100	MOVE
19	00010011	0000000000011000	0018	NULL
20	00010100	1000000000000000	8000	EAREG
21	00010101	0000010100000000	0500	JMP
22	00010110	00000000000 <u>1000</u>	0008	M1
23	00010111	XXXXXXXXXXXXXXXXXX		
24	00011000	0000000000000000	0000	
25	00011001	0000000000000001	0001	

Programm

Daten

Symbol	Binärcode	
MOVE	0000	0001
SUB	0000	0010
CMP	0010	0001
BNE	0000	1100
JMP	0000	0101



# Programm-Übersetzung (Assemblerierung)

---

Übersetzung kann „per Hand“ erfolgen. Dazu benötigt man:

- die Zuordnungstabelle für die OpCodes und
- die Adresse des ersten Befehls im Speicher, um die Symboltabelle mit den Adresszuordnungen aufstellen zu können.

**Nachteil:** sehr zeitaufwendig und fehleranfällig

Übersetzungsvorgang läuft jedoch nach festen Regeln ab:

➔ **Ausführung durch den Prozessor selbst**



# Programm-Übersetzung (Assemblerung)

---

## Assembler:

Programm, das einen Quellcode in Assemblersprache in eindeutiger Weise in Maschinencode übersetzt.

Die Regeln zur symbolischen Programmierung ergeben sich aus der Definition einer Assemblersprache

## Format einer Programmzeile:

Namensfeld (label)	Operationsfeld (OpCode)	Adreßfeld (Operanden)	Kommentarfeld
<i>symbolische Adressierung einer Programmzeile</i>	<i>symbolischer OpCode</i>	<i>symbolische Adreßangaben</i>	

# Assemblerdirektiven (Assembleranweisungen)

---

Assemblerdirektiven sind Befehle für den Assembler

- Erzeugung von Konstanten
- Wertzuweisung an Operanden
- Reservierung von Speicherplatz
- Steuerung des Assemblierungsvorgangs
- erzeugen bei der Übersetzung nicht immer Binärcode (im Gegensatz zu Maschinenbefehlen)
- sie unterliegen dem Format einer Programmzeile

# Assemblerdirektiven

---

[symbol]	ORG	c	origin of program or data
[symbol]	DS	c	define storage
[symbol]	DC	c	define constant
symbol	EQU	c	equate
	END		end of program

## Im Beispiel:

```
                ORG    0
EAREG    EQU    32678
NULL     DC     0
EINS     DC     1
ZEITK    DS     1
```

# Impulsgeber-Programm

Name	Opcode	Adreßangaben	Kommentar
*			
* Impulsgeberprogramm			
	ORG	0	Speicherbelegung ab Adresse 0
EAREG	EQU	32768	Ein-/Ausgabeadresse festlegen
	MOVE	EAREG,ZEITK	Zeitkonstante einlesen
	MOVE	NULL,EAREG	
	MOVE	NULL,R0	
M1	MOVE	ZEITK,R2	Zeitschleife initialisieren
M2	SUB	EINS,R2	
	CMP	R0,R2	Zeitbedingung abfragen
	BNE	M2	
	MOVE	EINS,EAREG	positive Flanke
	MOVE	NULL,EAREG	negative Flanke
	JMP	M1	
*			Beginn des Datenbereichs
ZEITK	DS	1	
NULL	DC	0	
EINS	DC	1	
	END		

# Impulsgeber- Programmliste

Nr.	Adresse	Inhalt	Name	Opcode	Adreßangaben	Kommentar
1			*			
2			* Impulsgeberprogramm			
3						
4				ORG	0	speichern ab Adr. 0
5			EAREG	EQU	32768	E/A-Adr. festlegen
6	0000	0100 8000 0017		MOVE	EAREG,ZEITK	Zeitkonst. einlesen
7	0003	0100 0018 8000		MOVE	NULL,EAREG	
8	0006	0108 0018		MOVE	NULL,R0	
9	0008	010A 0017	M1	MOVE	ZEITK,R2	Zeitschleife init.
10	000A	020A 0019	M2	SUB	EINS,R2	
11	000C	218A		CMP	R0,R2	Zeitbed. abfragen
12	000D	0C00 000A		BNE	M2	
13	000F	0100 0019 8000		MOVE	EINS,EAREG	positive Flanke
14	0012	0100 0018 8000		MOVE	NULL,EAREG	negative Flanke
15	0015	0500 0008		JMP	M1	
16			*			Datenbereichsanfang
17	0017		ZEITK	DS	1	
18	0018	0000	NULL	DC	0	
19	0019	0001	EINS	DC	1	
20				END		



# Assemblierung

---

## Zwei Phasen:

### 1. Phase:

- Adresszuordnung herstellen
- Fehlerliste anfertigen

### 2. Phase:

- Maschinencode erzeugen
- Symbolische und binäre Auflistung

Beim Erreichen der END-Anweisung müssen alle Symbole definiert sein!

# 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

# Warum MIPS?

---

**MIPS** (*machine with no interlocked pipe stages*)

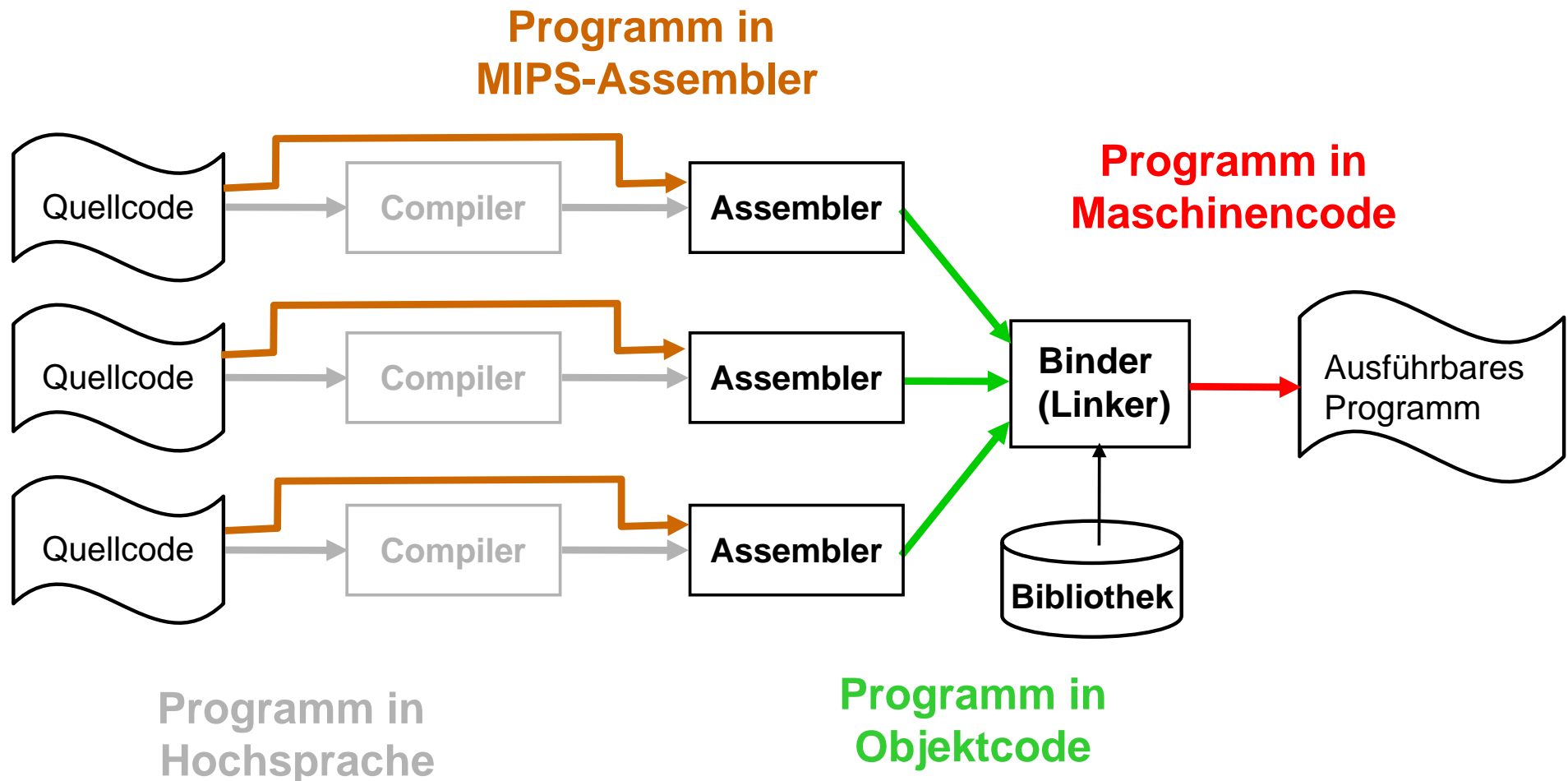
≠

**MIPS** (*million instructions per second*)

- Der MIPS R2000 ist ein sehr klar strukturierter RISC Prozessor
- Sauberer und klarer Befehlssatz
- Basis der richtungsweisenden Bücher von Hennessy/Patterson
- MIPS außerhalb von PCs (bei Druckern, Routern, ..) weit verbreitet
- SPIM-Simulator verfügbar
  - Keine Gefährdung des laufenden Systems
  - Bessere Interaktionsmöglichkeiten
  - SPIM läuft auf PCs, und Workstations (Linux/Sun-OS, MacOS, Windows)

# Der SPIM-Simulator

Der SPIM-Simulator ist Assembler, Linker und Debugger in einem Programm.



# Installation und Benutzung

---

- ❑ Simulator abrufbar unter:

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

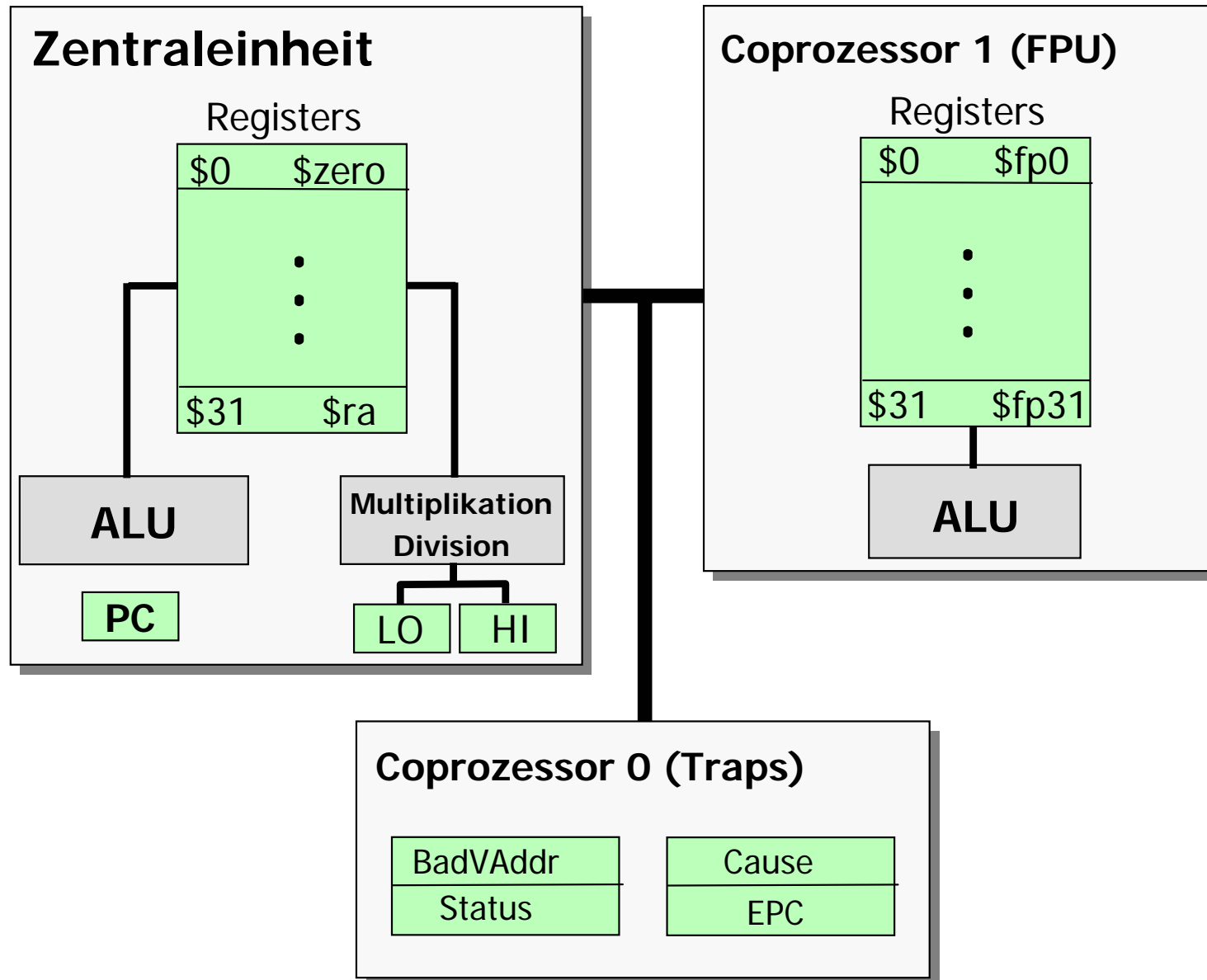
- ❑ SPIM unterliegt nur der GNU-Lizenz. Die neueste Version ist erhältlich unter:

**`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



# Koprozessoren

---

Der MIPS-Prozessor besitzt zwei Koprozessoren

- ❑ **Coprocessor 0 (Traps):**

Register enthalten Informationen über den Prozessorstatus und die Ursachen von Unterbrechungen und Ausnahmen

- ❑ **Coprocessor 1 (FPU) für Gleitkomma-Arithmetik:**

Enthält 32 allgemein verwendbare 32-Bit Gleitkomma-Register

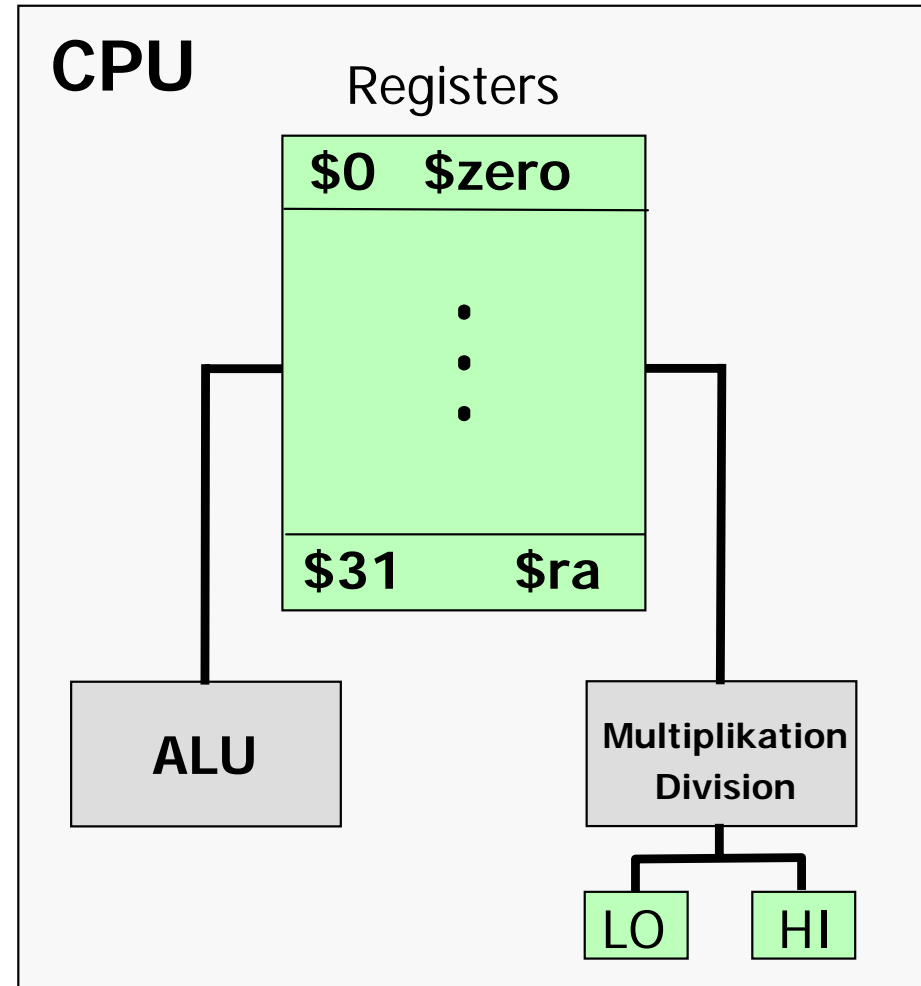
# Registersatz

## 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 verwendbare Register.

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



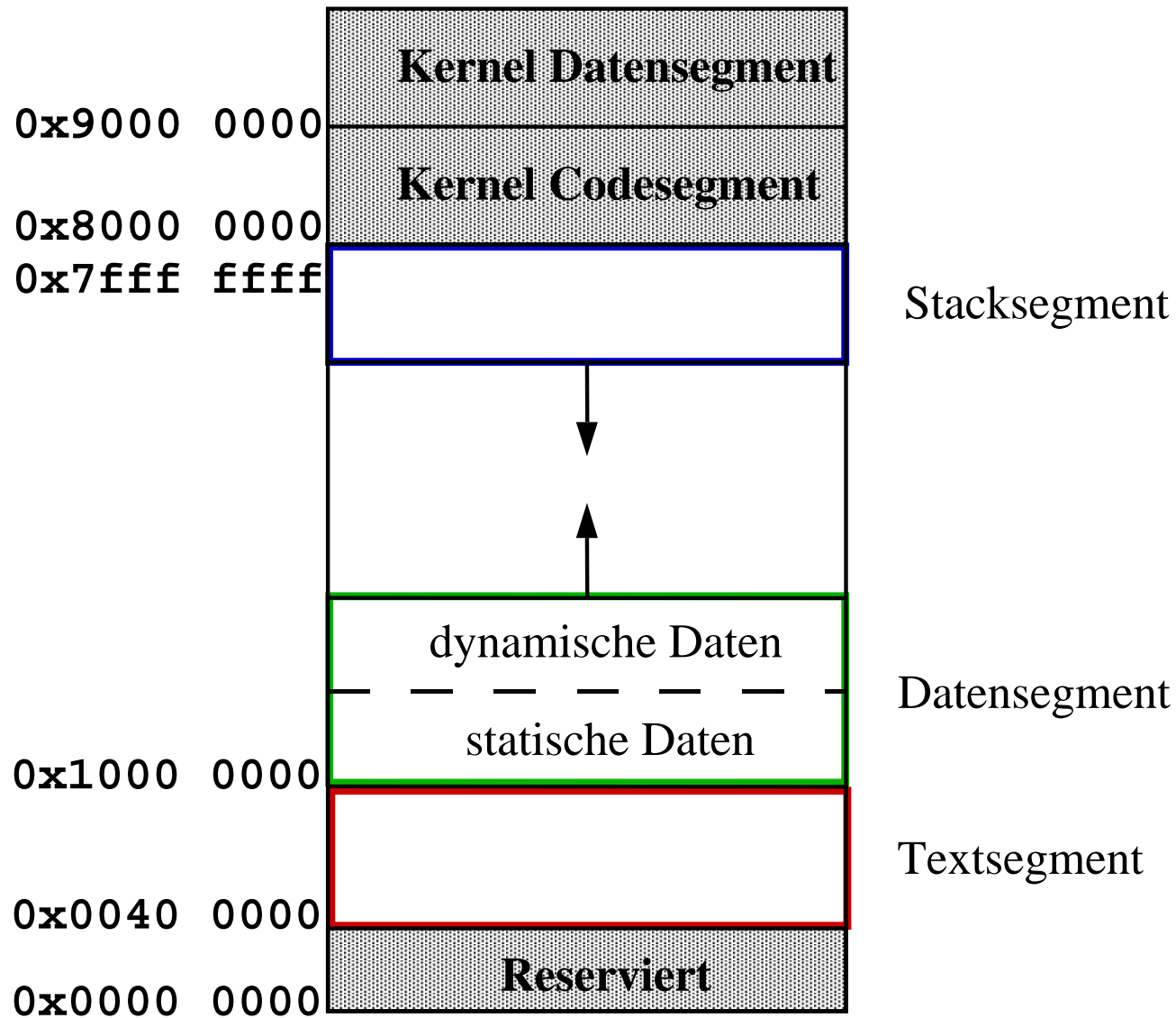
# Registersatz

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:           .ascii "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
```

<code>print_string</code>	<code>4</code>	Adresse des String in \$a0
---------------------------	----------------	----------------------------

# Ausgabe einer Integerzahl mit CR

---

```
                .data

cr_string:      .asciiz "\n"

                .text

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

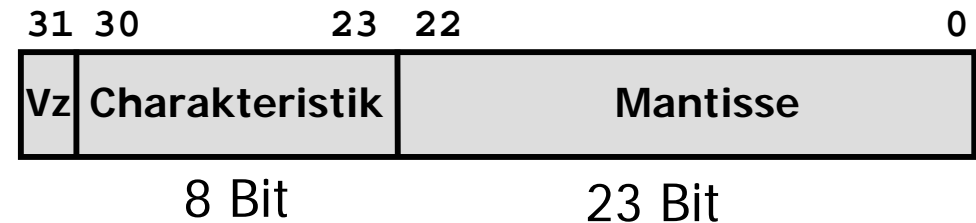
# Datenformate im MIPS-Prozessor

---

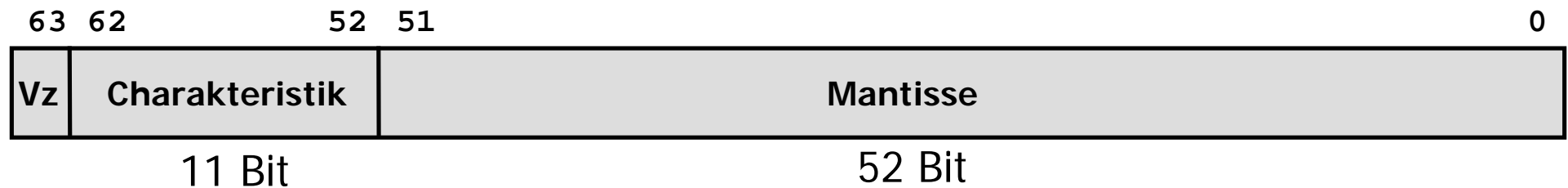
- ❑ Es sind folgende Datenformate definiert:
  - Byte (8 Bit), Halbwort (16 Bit), Wort (32 Bit)
- ❑ Ordnung von Bytes in Wörtern und Wörter in mehrfachen Wortstrukturen
  - Little endian order oder
  - Big endian order
- ❑ Vorzeichenbehaftet Zahlen werden in Zweierkomplement-Form dargestellt
- ❑ Ganze 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

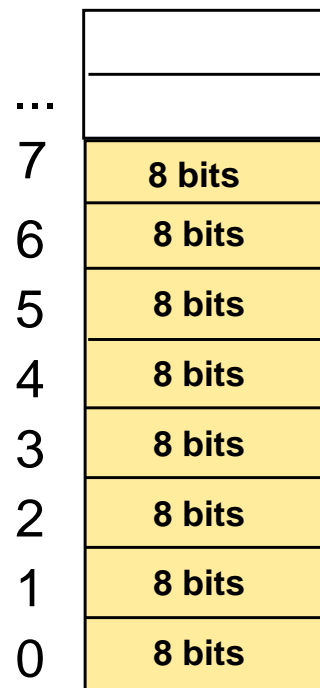
Es werden Wörter geladen, aber Bytes adressiert.

- $2^{32}$  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.

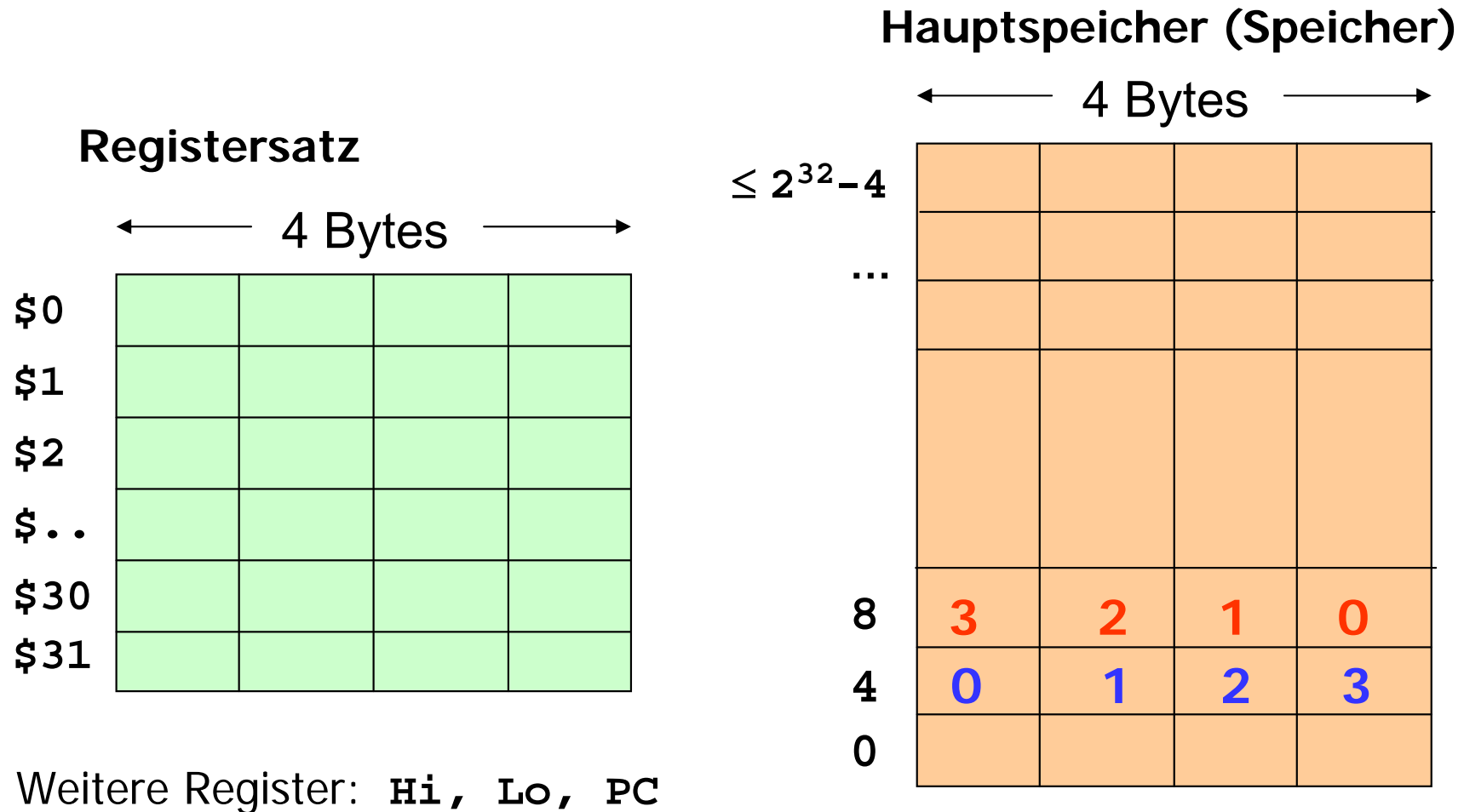
**Welche Werte haben die 2 niedrigstwertigen Bits einer Wort-Adresse?**

# 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.



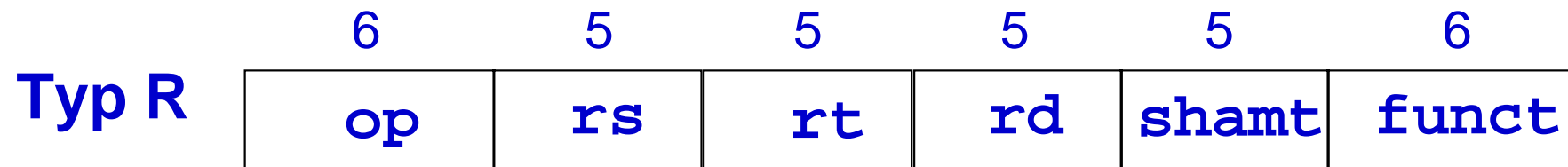
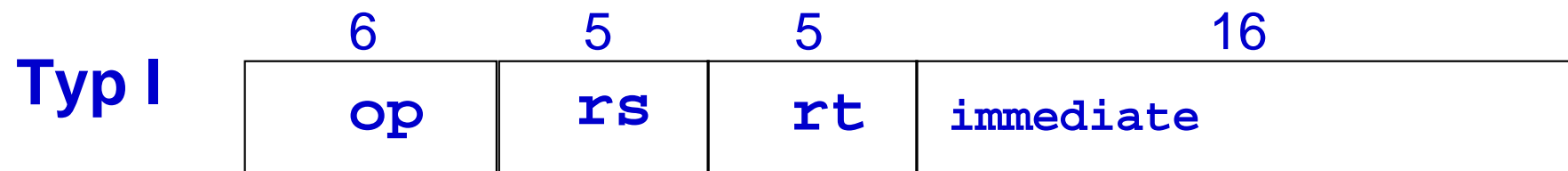
# Das Speichermodell der MIPS-Architektur



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.

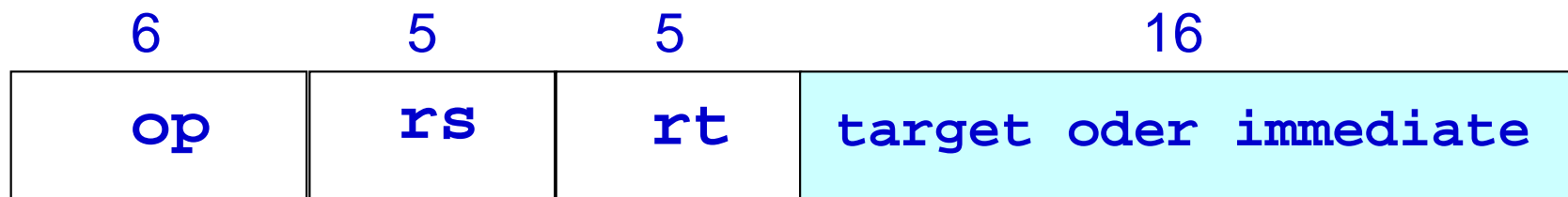
# Adressierungsarten des MIPS-Prozessors

---

## Registeradressierung: (register)

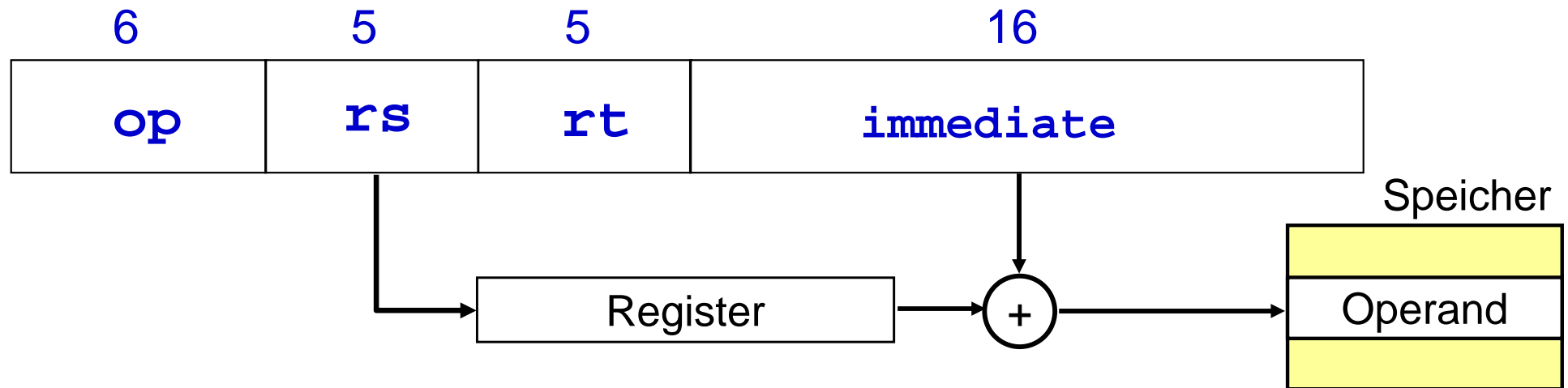


## Direkte Adressierung: imm

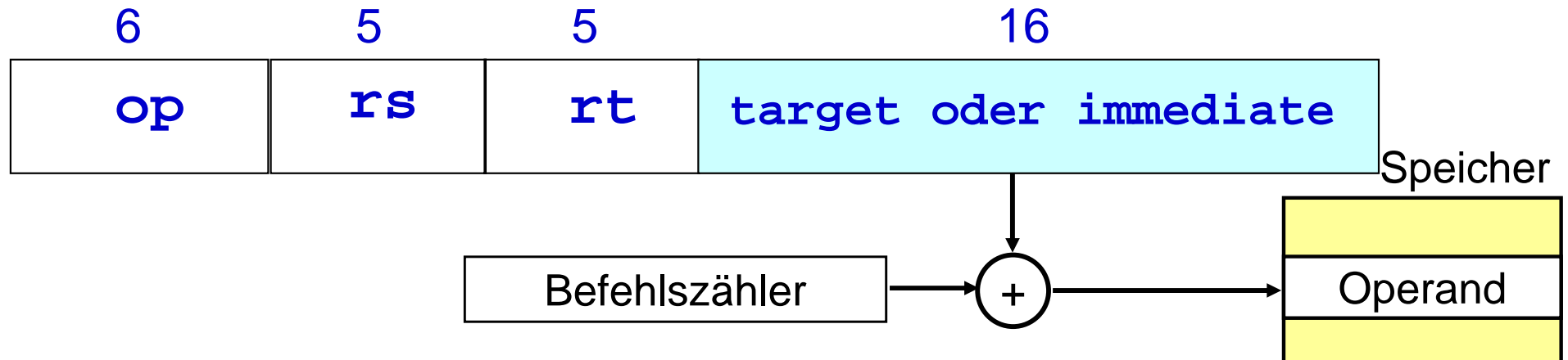


# Adressierungsarten des MIPS-Prozessors

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



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



# Adressierungsarten in SPIM

Format	Beispiel	Adressberechnung
(register)	( \$s0 )	Inhalt des Registers
imm	0x10003248	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

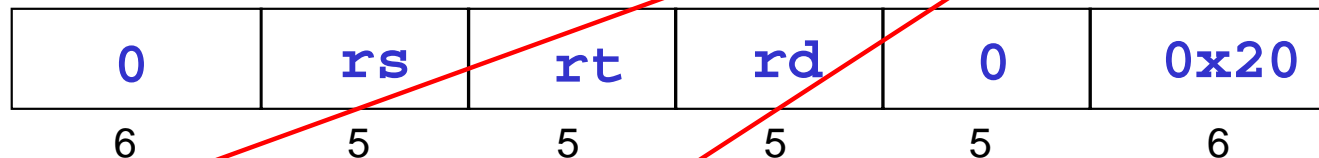
## Arithmetische Befehle

- 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**

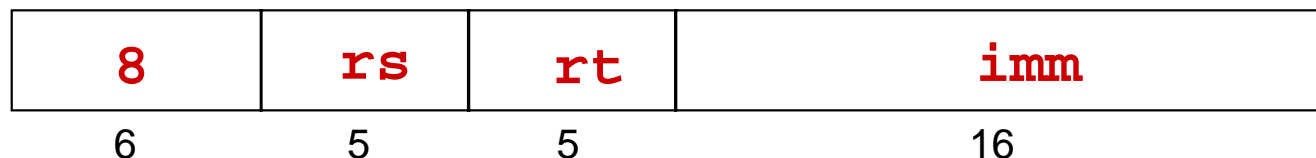


Zielregister

**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
```

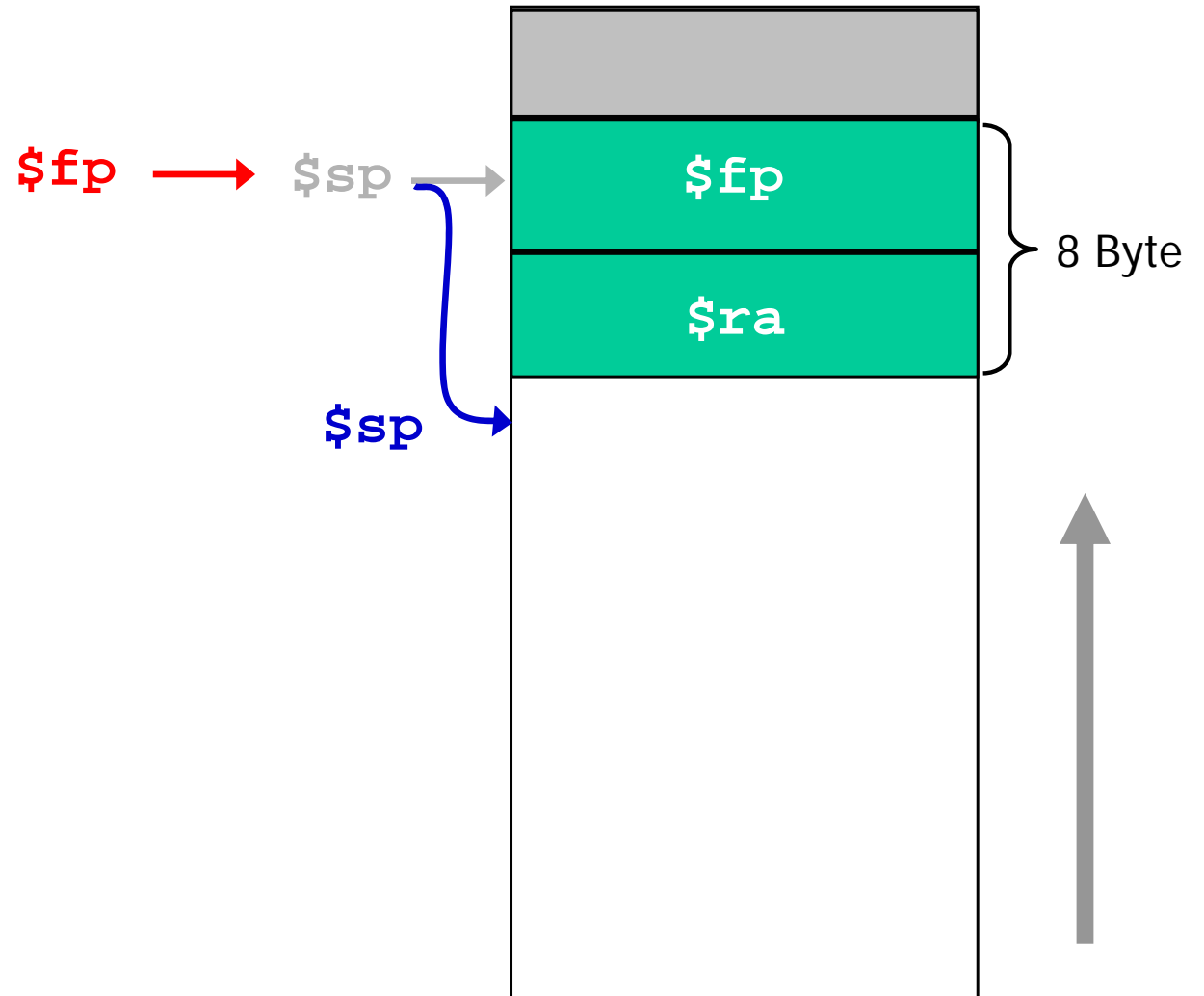
# Struktur eines MIPS-Programms

```
.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:    .asciiz "\n"           # Sonderzeichen "neue Zeile"
eingabeA:     .asciiz "Integer-Zahl A: "
eingabeB:     .asciiz "Integer-Zahl B: "
result_sum:   .asciiz "A + B = "
result_dif:   .asciiz "A - B = "
result_mul:   .asciiz "A * B = "
result_div:   .asciiz "A mod B = "
result_rst:   .asciiz "Rest = "
error_str:    .asciiz "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
```