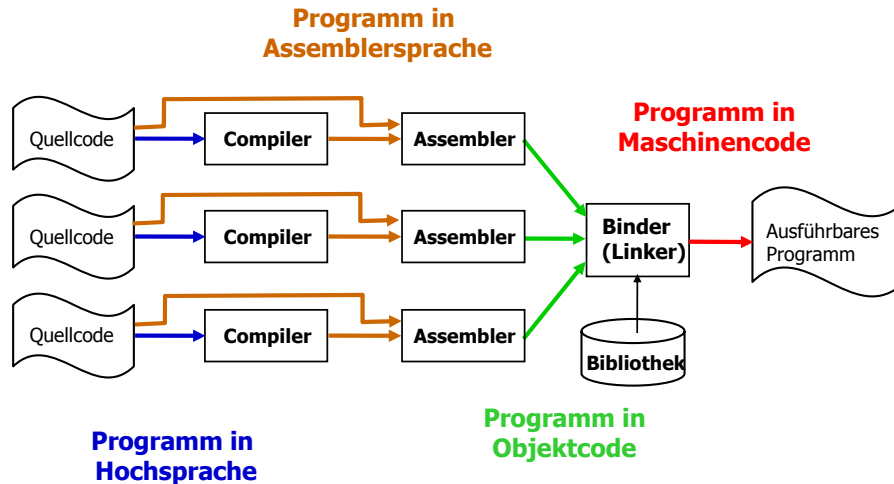


## Vom Quellcode zum ausführbaren Programm



## Vom Quellcode zum ausführbaren Programm

### □ **Assembler:**

Programm, das einen Quellcode in Assemblersprache in eindeutiger Weise in Maschinsprache 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 mehrere Objektcode-Module auflöst und sie zu einem ausführbaren Programm verbindet

## Hochsprachen- und Assemblerprogramme

### Warum in Assemblersprache programmieren?

- zeit- und sicherheitskritische Programmteile
- vollständiger Zugriff auf die Hardware
- Reduktion der Größe des Programms
- Verständnis für die Funktionsweise eines  $\mu P$
- Verständnis der Funktionsweise von Compilern

### Nachteile:

- sehr aufwendige und fehleranfällige Programmierung
- umfangreicher und schlecht wartbarer Quellcode
- maschinenspezifisch

## Hochsprachen- und Assemblerprogramme

### Beispiele für Hochsprachen:

- c (nahe an der Hardware), C++ (objektorientierte Erweiterung von c)
- VHDL (Hardwarebeschreibung)
- ADA (Echtzeitsysteme, eingebettete Systeme)
- JAVA, LISP, ...

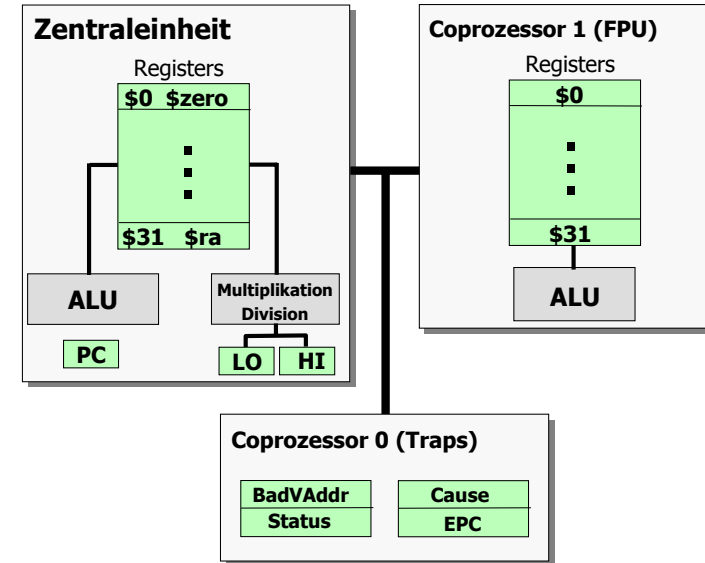
### Beispiele für Assemblersprachen:

- Spezifisch für ganze Prozessorfamilien
- MIPS-Assembler, SPARC-Assembler, VAX-Assembler, 86000-Assembler, 8086-Assembler

# MIPS-Assembler

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

# Aufbau des MIPS-Prozessors



# Koprozessoren

Der MIPS-Prozessor besitzt zwei Koprozessoren

Behandlung von Unterbrechungen und Ausnahmen  
Fließkomma-Arithmetik

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

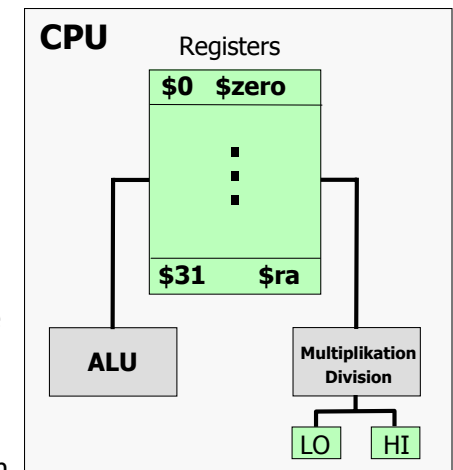
# Registersatz

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.



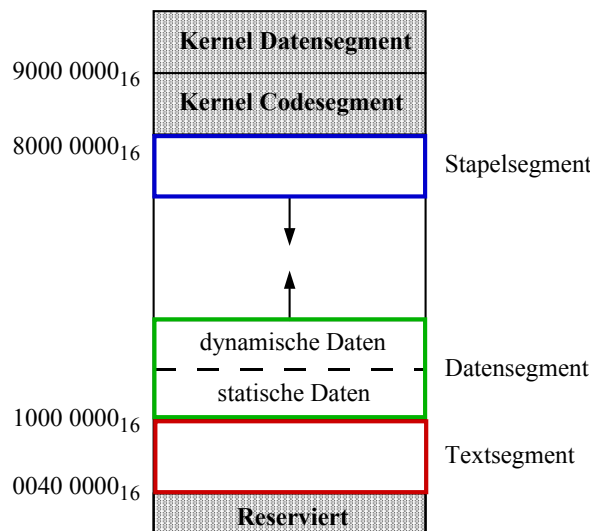
# Registersatz

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 Rücksprungsadresse 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)*)
- ❑ **Stapelsegment:** beinhaltet lokale Daten und Rücksprungsadressen für Unterprogramme
- ❑ **Kernelsegmente:** beinhalten betriebssystemeigene Daten und Prozeduren

# Syntax der MIPS-Assemblersprache

Ein Assemblerprogramm besteht aus

- Assemblerdirektiven
- Marken (Labels)
- Maschinenbefehlen
- 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**

## Assemblerdirektiven

Assemblerdirektiven sind Befehle für den Assembler:

- Erzeugung von Konstanten
- Wertzuweisung an Operanden
- Statische Reservierung von Speicherplatz
- Platzierung vom Programmcode und Daten
- Steuerung des Assemblierungsvorgangs

Sie erzeugen bei der Übersetzung nicht immer Binärcode (im Gegensatz zu Maschinenbefehlen)

Sie unterliegen dem Format einer Programmzeile

## 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 Byte 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 b1, ..., bn`

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

`.half h1, ..., hn`

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

`.word w1, ..., wn`

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

`.float f1, ..., fn`

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

## MIPS-Assemblerdirektiven

`.double d1, ..., dn`

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

```
.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 0,0,0,0                     # 4 Integers ...

.text                                     # Es folgt der Programmcode

.globl main                               # main ist globales Symbol
main:   lw $t0, PI
```

# Adressraumorganisation

## ➤ Adressraum

- **byte-adressierbar**, d. h. jedes Byte in einem Speicherwort kann einzeln adressiert werden.
- **wort-adressierbar**, d. h. nur 16-, 32- oder 64-Bit-Wörter können direkt adressiert werden.

## ➤ In der Regel muss der Zugriff auf Speicherwörter **ausgerichtet** (aligned) sein.

- Ein Zugriff zu einem Speicherwort mit einer Länge von n Bytes ab der Speicheradresse A heißt ausgerichtet, wenn  $A \bmod n = 0$  gilt.

# Byteanordnung im Speicher

- Das **Big Endian Format** („most significant byte first“) speichert von links nach rechts, d.h. die Adresse des Speicherwortes ist die Adresse des höchstwertigen Bytes des Speicherwortes.

Wortadresse	x0				x4			
Bytestelle im Wort	7	6	5	4	3	2	1	0

- Das **Little Endian Format** („least significant byte first“) speichert von rechts nach links, d.h., die Adresse eines Speicherwortes ist die Adresse des niedrigstwertigen Bytes des Speicherwortes

Wortadresse	x0				x4			
Bytestelle im Wort	0	1	2	3	4	5	6	7

# Adressierungsarten

## Adressierungsarten:

die verschiedenen Möglichkeiten eines Prozessors die Adresse eines Operanden oder eines Sprungziels im Speicher zu berechnen

**Früher:** Adresse der Operanden und Sprungziele absolut im Befehl vorgegeben

## Nachteile:

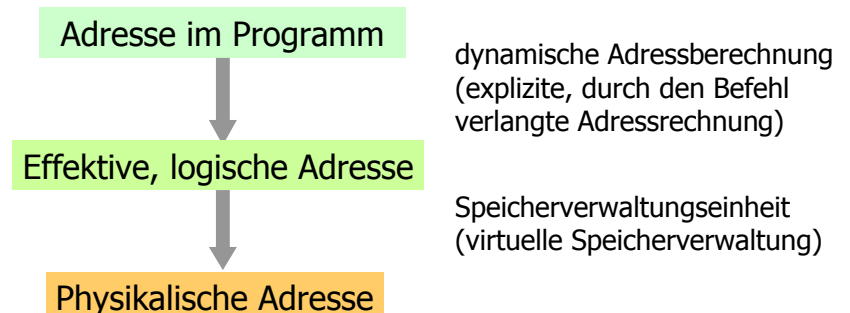
- absolute Adressen müssen bereits zur Programmierzeit festgelegt werden → Programme sind lageabhängig im Speicher
- Bei Tabellenzugriffen im Speicher muss die Adresse im Befehl geändert werden → keine Festwertspeicher als Programmspeicher möglich

# Adressierungsarten

## Abhilfe:

Adresse wird zur Laufzeit berechnet (dynamische Adressberechnung)

## Ablauf der Adressberechnung:



## Klassen der Adressierungsarten

### □ Register-Adressierung

Operand steht bereits in einem Register → kein Speicherzugriff erforderlich

### □ Einstufige Speicher-Adressierung

Eine Adressberechnung zur Ermittlung der effektiven Adresse notwendig, d.h. keine mehrfachen Speicherzugriffe zur Adressermittlung

### □ Zweistufige Speicher-Adressierung

Mehrere sequentielle Adressberechnungen und Speicherzugriffe. Ergebnis der ersten Berechnung liefert die Adresse einer Speicherzelle, deren Inhalt wieder eine Adresse oder ein Offset zur weiteren Berechnung ist



## Klassen der Adressierungsarten

Visualisierung der ersten beiden Klassen auf der TI-Homepage

<http://i61www.ira.uka.de/users/asfour/TI/Adressierungsarten/>



## Adressierungsarten des MIPS-Prozessors

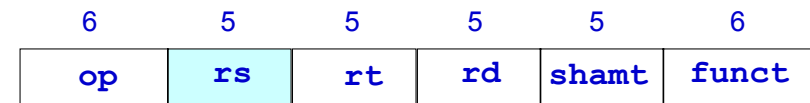
Der MIPS-Prozessor unterstützt vier Adressierungsarten:

- **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.
- **Programmzähler-relative Adressierung:** Die Adresse ergibt sich aus dem Wert des Programmzählers und einer Konstanten im Befehl.

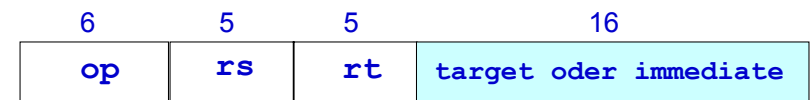


## Adressierungsarten des MIPS-Prozessors

### Registeradressierung: (register)

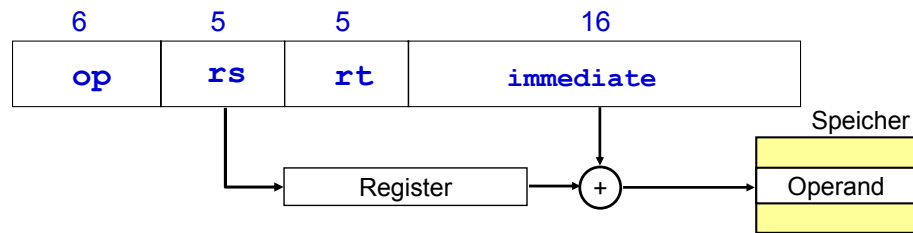


### Direkte Adressierung: imm

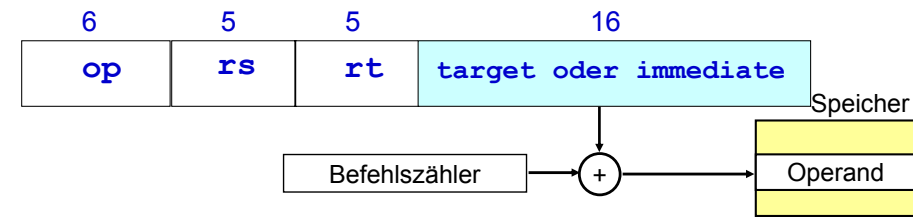


# Adressierungsarten des MIPS-Prozessors

## Basisadressierung: imm(register)

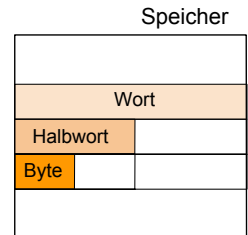


## Befehlszähler-relative Adressierung: imm(PC)



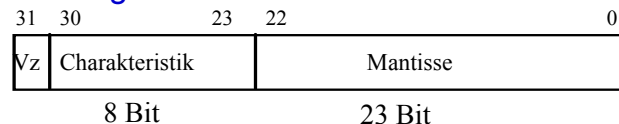
# Datenformate im MIPS-Prozessor

- Es sind folgende Datenformate definiert:
  - Byte (8 Bit), Halbwort (16 Bit), Wort (32 Bit)
- Ordnung von Bytes in Wörtern und Wörter in mehrfachen Wortstrukturen wird die **Big Endian** Konvention verwendet:
  - Adresse eines Wortes ist die Adresse des "most significant" Bytes
  - Ein Wort wird mit der Byteadresse seines höchstwertigen Bytes adressiert.
- Vorzeichenbehaftet Zahlen werden in Zweierkomplement Form dargestellt
- Ganze Zahlen werden entweder vorzeichenlos (unsigned) oder vorzeichenbehaftet (signed) in Zweierkomplement Form dargestellt

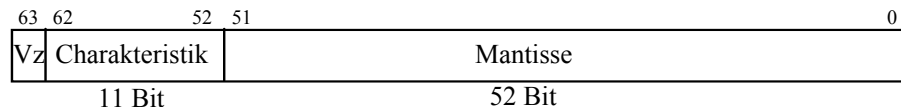


# Fließkommaformate

## Einfache Genauigkeit:



## Doppelte Genauigkeit:



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

# Befehlssatz

## Welche Operationen können auf die Daten ausgeführt werden?

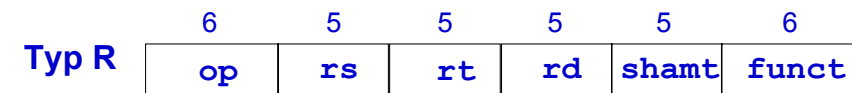
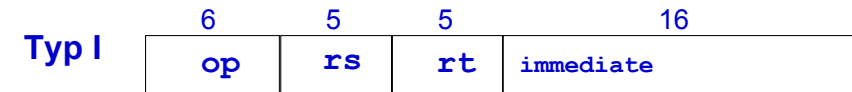
- Der **Befehlssatz** (*instruction set*) legt die Grundoperationen eines Prozessors fest.
- Befehlsarten:**
  - Datenbewegungsbefehle
  - Arithmetisch-logische Befehle
  - Schiebe- und Rotationsbefehle
  - Multimediabefehle
  - Gleitkommabefehle
  - Programmsteuerbefehle
  - Systemsteuerbefehle
  - Synchronisationsbefehle

# Befehlsformate

- Das **Befehlsformat** (*instruction format*) definiert, wie die Befehle codiert sind.
- Eine **Befehlscodierung** beginnt mit dem **Opcode**, der den Befehl selbst festlegt.
- In Abhängigkeit vom Opcode werden weitere Felder im Befehlsformat benötigt.
- Art der **Adressformate** definiert vier Klassen von **Befehlssätzen**:
  - **Dreiadressformat:** Opcode Dest Src1 Src2
  - **Zweiadressformat:** Opcode Dest/Src1 Src2
  - **Einadressformat:** Opcode Src
  - **Nulladressformat:** Opcode

# Befehlsformate

Der MIPS-Prozessor hat ausschließlich Befehle feste 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
rs	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

# 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

## Befehlssatz

Befehlsformate (z. B. bei der Addition):

**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

## Befehlssatz

### logische Befehle

- logisches AND      `and rd,rs,rt`    `andi rd,rs,imm`
- logisches NOR      `nor rd,rs,rt`
- logische Invertierung    `not,rdest,rsrc`
- logisches XOR      `xor rd,rs,rt`    `xori rd,rs,imm`
- logisches OR      `or rd,rs,rt`    `ori rd,rs,imm`
- bitweise Rotieren      `rol/ror rdest,rsrc1,rsrc2`
- bitweise Schieben      `sll rd,rs,rt` (rt = distance)  
                              `sllv rd,rs,rt`  
                              `sra rd,rs,rt` (rt = distance)  
                              `srlv rd,rs,rs`

## Befehlssatz

### Befehle zum Laden von Konstanten

- Laden einer Konstante in ein Register  
    `li rdest,imm`                      `lui rdest, imm`

### Vergleichsbefehle

- Vergleich zweier Register  
    `slt/sltu rd,rs,rt`      `slti/sltiu rd,rs,imm`  
    `seq rdest,rsrc1,rsrc2`  
    `sge/sgeu rdest,rsrc1,rsrc2`  
    `sgt/sgtu rdest,rsrc1,rsrc2`  
    `sle/sleu rdest,rsrc1,rsrc2`  
    `sne rdest, rsrc1, rsrc2`
- Vergleich eines Registers mit Null

## Befehlssatz

### Kontrollflussbefehle

- unbedingtes Verzweigen zu einer Adresse  
    `j target`      `b label`
- unbedingtes Verzweigen zu einer Adresse und sichern der nachfolgenden Adresse (für Unterprogramme)  
    `jal target`
- Verzweigen, wenn Bedingungs flag eines Coprozessors wahr/falsch ist  
    `bczt label`    `bczf label`
- Verzweigen, wenn ein Register größer/kleiner als ein anderes Register ist  
    `bgt rsrc1,rsrc2,label`      `bgtu rsrc1,rsrc2,label`  
    `blt rsrc1,rsrc2,label`      `bltu rsrc1,rsrc2,label`  
    (auch `bge`, `bgeu`, `ble`, `bleu`)

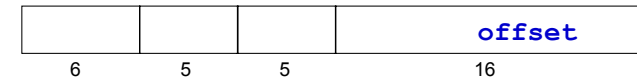
## Befehlssatz

### Kontrollflussbefehle

- Verzweigen, wenn zwei Register gleich/ungleich sind  
`beq rs,rt,label`      `bnq rs,rt,label`
- Verzweigen, wenn ein Register größer/kleiner als Null ist  
`bgtz rs, label`      `bltz rs,label`  
(auch `bgez, blez`)
- Verzweigen, wenn ein Register gleich/ungleich Null ist  
`beqz rsrc, label`      `bnez rs,label`

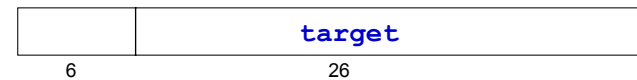
## Befehlssatz

### Branch



Offset: 16 <sup>11</sup>, vorzeichenbehaftet  
→  $2^{15}$  1Befehle vorwärts und  $2^{15}$  rückwärts

### jump



26 <sup>11</sup> Adress- Feld

## Befehlssatz

### Lade- und Speicherbefehle

- Laden einer Adresse  
`la rdest, address`
- Laden und Speichern eines Bytes, Halbwortes, Wortes und Doppelwort  
`lb rt, address`      `sb rt, address`  
`lbu rt, address`  
`lh rt, address`      `sh rt, address`  
`lhu rt, address`  
`lw rt, address`      `sw rt, address`  
`ld rt, address`      `sd rt, address`
- Laden und Speichern von Koprozessor- Registern  
`lwcx rt, address`      `swcx rt, address`      (z=1, FPU)

## Befehlssatz

### Lade- und Speicherbefehle:

- Laden und Speichern von/an nicht ~~aus~~gerichtete Adressen  
`lwl rt, address`  
`lwr rt, address`  
`ulh rdest, address`      `ush rsrc, address`  
`ulhu rdest, address`  
`ulw rdest, address`      `usw rsrc, address`  
`ulhu rdest, address`

## Befehlssatz

### Befehle zum Verschieben von Daten:

- Verschieben eines Registerinhalts in ein anderes Register  
`move rdest, rsrc`
- Laden des Registers HI oder LO in ein allgemeines Register  
`mfhi rd`                      `mflo rd`  
`mthi rs`                      `mtlo rs`
- Verschieben von/nach Registern in Koprozessoren  
`mfcz rt, rd`                      `mtcz rd, rt`  
`mfc1 rt, rd`                      `mtc1 rd, rt`  
  
`mfc1.d rdest, frsrc1`  
 (frsrc1 und frsrc1+1 in die CPU-Register rdest und rdest+1)



## Befehlssatz

### Befehle für Fließkomma-Arithmetik:

- Arithmetik  
`abs.d fd, fs`                      `abs.s fd, fs`  
`add.d fd, fs, ft`                      `add.s fd, fs, ft`  
`sub.d fd, fs, ft`                      `sub.s fd, fs, ft`  
`mul.d fd, fs, ft`                      `mul.s fd, fs, ft`  
`div.d fd, fs, ft`                      `div.s fd, fs, ft`
- Vergleiche  
`c.eq.d fs, ft`                      `c.eq.s fs, ft`  
`c.le.d fs, ft`                      `c.le.s fs, ft`  
`c.lt.d fs, ft`                      `c.lt.s fs, ft`
- Laden und Speichern  
`l.d fdest, address`                      `l.s fdest, address`  
`s.d fdest, address`                      `s.s fdest, address`



## Befehlssatz

### Befehle für die Unterbrechungs- und Ausnahmebehandlung:

- Wiederherstellen des Status Registers nach einer Unterbrechung  
`rfe`

0x10	1	0	0x20
6	1	19	6
- Systemaufruf  
`syscall`

0	0	0xc
6	20	6
- Software-Unterbrechung  
`break code`

0	code	0xd
6	20	6
- Dummy-Operation  
`nop`

0	0	0	0	0	0
6	5	5	5	5	6

