

3. Übung

□ MIPS-Assembler

- SPIM-Simulator
 - Programmiermodell (Koprozessoren, Registersatz)
- Syntax der MIPS-Assemblersprache
- Assemblerdirektiven
- Eingabe und Ausgabe (SPIM-Systemaufrufe)
- Hello MIPS-World
- Adressierungsarten, Befehlsformate
- Befehlssatz



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:    ...
```



Systemaufrufe

Die Nummer des Systemaufrufes wird in 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:  .asciiz "\n"

        .text

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



Struktur eines MIPS-Programms

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

        .globl main
main:    subu $sp, $sp, 8           # Stack Frame ist 8 Bytes
        sw $ra, 0($sp)            # Sichern der Ruecksprungadresse
        sw $fp, 4($sp)            # Sichern des alten Frame-Pointers
        addu $fp, $sp, 8          # neuen Frame-Pointer definieren

# Hauptprogramm

        lw $ra, 0($sp)            # Ruecksprungadresse wiederherstellen
        lw $fp, 4($sp)            # Frame-Pointer wiederherstellen
        addu $sp, $sp, 8          # Stack-Frame loeschen
        jr $ra
```



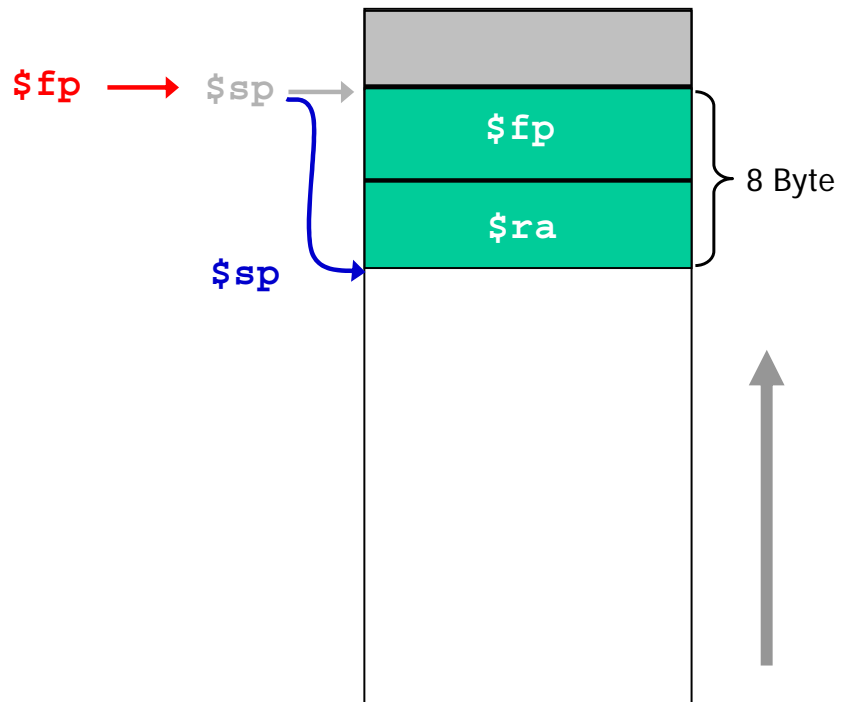
Struktur eines MIPS-Programms

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

.globl main
main: subu $sp, $sp, 8
      sw $ra, 0($sp)
      sw $fp, 4($sp)
      addu $fp, $sp, 8

      # Hauptprogramm

      lw $ra, 0($sp)
      lw $fp, 4($sp)
      addu $sp, $sp, 8
      jr $ra
```



Adressierungsarten des MIPS-Prozessors

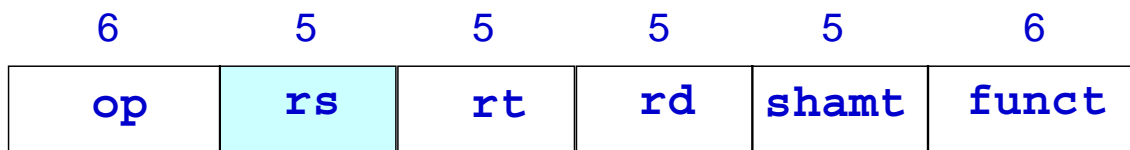
Der MIPS-Prozessor unterstützt vier Adressierungsarten:

- ❑ **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 Programmzählers und einer Konstanten im Befehl.

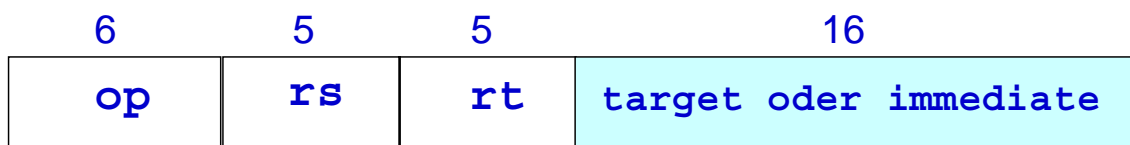


Adressierungsarten des MIPS-Prozessors

Registeradressierung: (register)

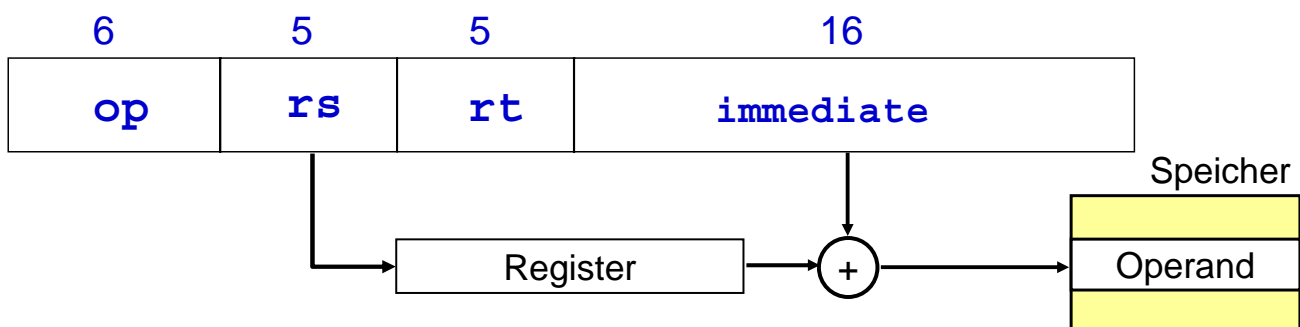


Direkte Adressierung: imm

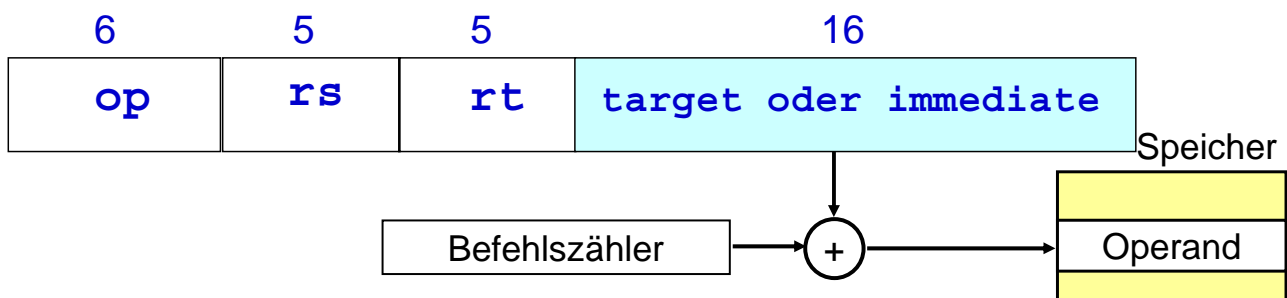


Adressierungsarten des MIPS-Prozessors

Basisadressierung: imm(register)



Befehlszähler-relative Adressierung: imm(PC)



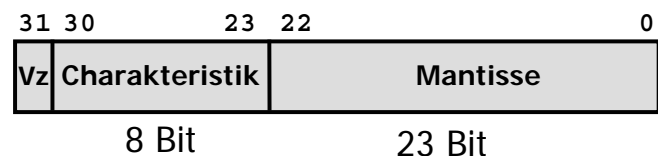
Datenformate im MIPS-Prozessor

- ❑ Es sind folgende Datenformate definiert:
 - Byte (8 Bit), Halbwort (16 Bit), Wort (32 Bit)
- ❑ Ordnung von Bytes in Wörtern und Wörter in mehrfachen Wortstrukturen
 - 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

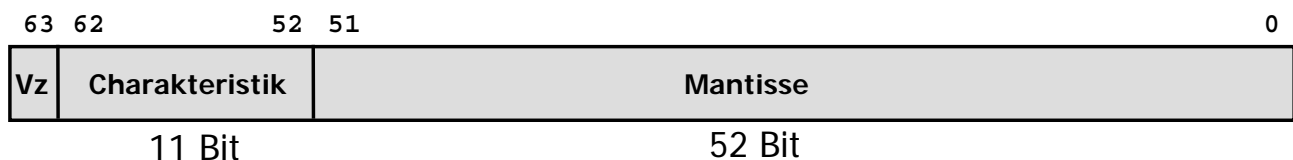


Fließkommaformate

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?



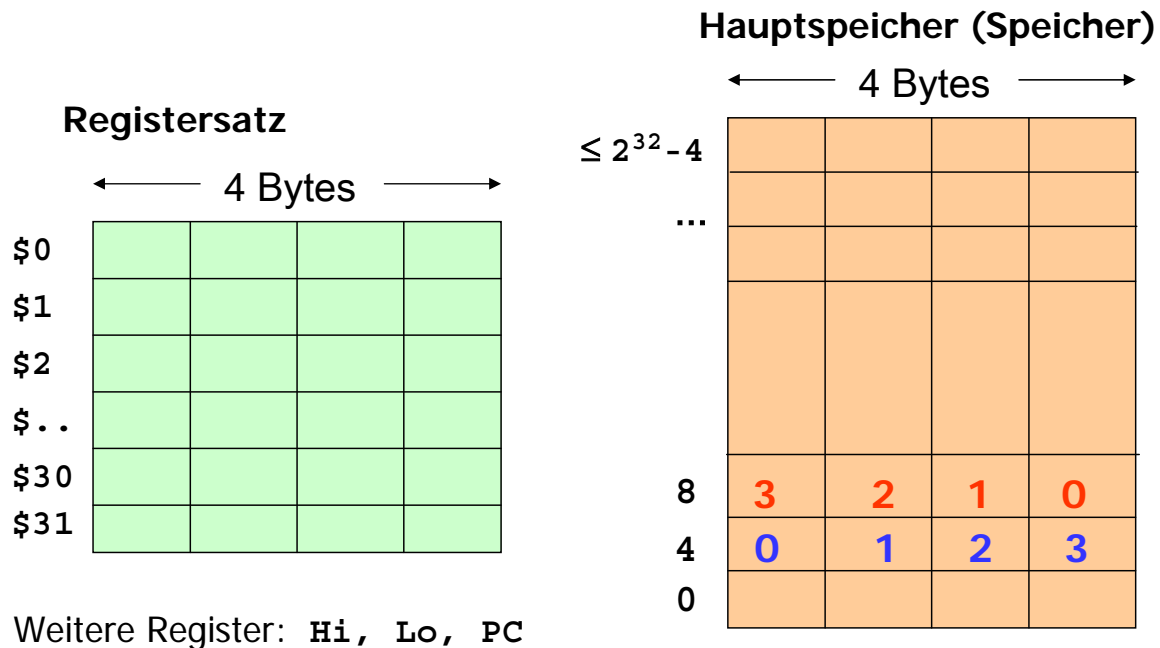
Das Speichermodell

- 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

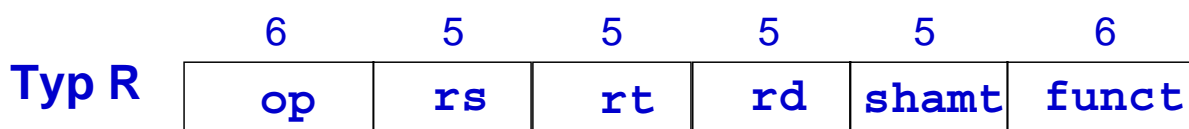
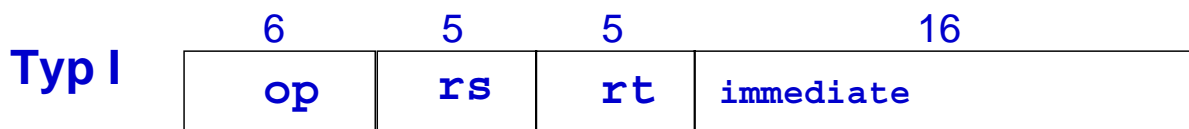


MIPS unterstützt
big und **little** endian order



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
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

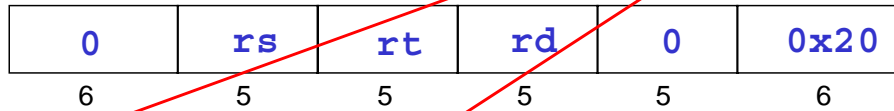
- Absolutwert:
`abs rdest, rsrc`
- Addition:
`add rd,rs,rt` `addu rd,rs,rt` `addi rd,rs,imm`
- Division (Quotient in LO und Rest in HI)
`div rs,rt` `divu rd,rs1,rs2`
- Multiplikation
`mult rs, rt` `multu rs, rt (unsigned)`
`mul rdest,rsrc1,rsrc2` `mulo rdest,rsrc1, rsrc2`
- Negation
`neg rdest,rsrc`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



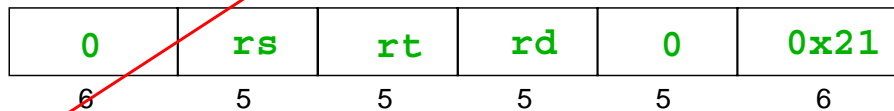
Beispiel: Additionsbefehle in MIPS

Befehlsformate (z. B. bei der Addition):

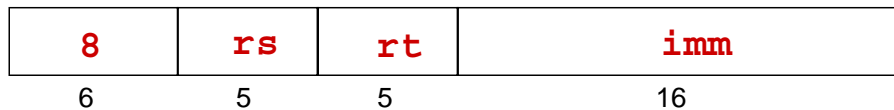
`add rd,rs,rt`



`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: `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`



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 $ra, 0($sp)      # Sichern der Ruecksprungsadresse
           sw $fp, 4($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 $ra, 0($sp)    # Ruecksprungadresse wiederherstellen
          lw $fp, 4($sp)   # Frame-Pointer wiederherstellen
          addu $sp, $sp, 8 # Stack-Frame loeschen
          jr $ra
```



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,imm` (`imm = distance`)
`sllv rd,rs,rt`
`sra rd,rs,imm` (`imm = 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



Vergleichsbefehle

```
if $s1 < $s2 then  
    $t0 = 1  
else  
    $t0 = 0
```

} `slt $t0, $s1, $s2`

`slt rd, rs, rt`

slt: set less than

0	rs	rt	rd	0	0x2a
6	5	5	5	5	6



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

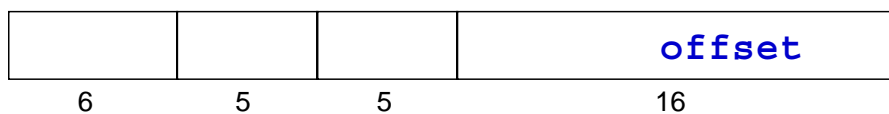
Kontrollflussbefehle

- 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`



Kontrollflussbefehle

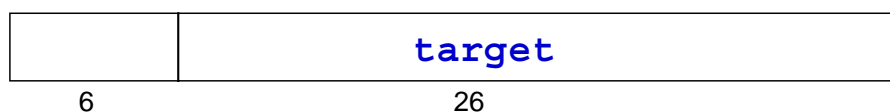
Branch



Offset: 16-bit, vorzeichenbehaftet

➔ $2^{15}-1$ Befehle vorwärts und 2^{15} rückwärts

jump



26-bit Adress-Feld



Kontrollflussbefehle

Einstufige Verzweigung:

Eine einstufige Verzweigung wird durch einen bedingten Sprung realisiert

if-Anweisung

```
if ( register_s1 == 0)
{
    if-Anweisungen
}
next-Teil;
```

Assembler-Code

```
beqz    $s1, marke1
j       marke2

marke1: { ..... } if-Anweisungen
        { ..... }
        { ..... }

marke2:  ....  next-Teil
```



Kontrollflussbefehle

Zweistellige Verzweigung:

if-else-Anweisung

```
if (register_s1 == 0)
{
    if-Anweisungen
}
else
{
    else-Anweisungen
}
next;
```

Assembler-Code

```
beqz    $s1, marke1
        { ..... } else-Anweisungen
        { ..... }
        { ..... }

j       marke2

marke1: { ..... } if-Anweisungen
        { ..... }
        { ..... }

marke2:  ....  next-Teil
```



Kontrollflussbefehle

Bedingte Verzweigung

```
bne $t0, $t1, Label
```

```
beq $t0, $t1, Label
```

```
if (i==j)
    h = i + j;
```

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:    ....
```



Kontrollflussbefehle

Unbedingte Verzweigung

```
j label
```

```
if (i!=j)
    h=i+j;
else
    h=i-j;

        beq $s4, $s5, Lab1
        add $s3, $s4, $s5
        j Lab2
Lab1:  sub $s3, $s4, $s5
Lab2:  ...
```



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-ausgerichtete 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`

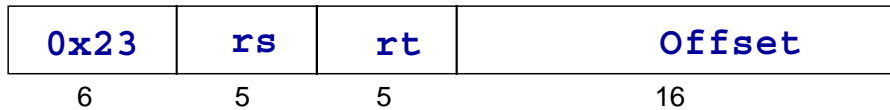


Lade-und Speicherbefehle

- Laden/Speichern von Bytes, Halbwörter und Wörter

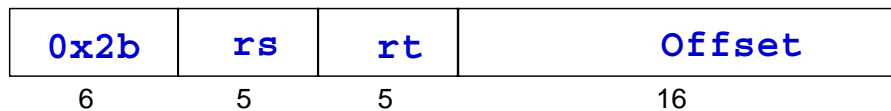
lw rt, address

Lade das 32-Bit Wort an der Adresse **address** ins Register **rt**



sw rt, address

Speichere das 32-Bit Wort im Register **rt** an der Adresse **address**



Beispiel

Lade- und Speicher-Befehle :

C-Code: `A[8] = h + A[8];`

MIPS code: `lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)`



Unterscheid zwischen **lb** und **lbu**

```
.data
result: .word 0x89abcdef 0x79abcdef

.text

# Start des Hauptprogrammes

.globl main
main:
...
lbu $a0, result
# $a0 enthaelt 0x00000089 0x00000079
...
lb $a0, result
# $a0 enthaelt 0xffffffff89 0x00000079
...
jr $ra
```



MIPS-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
```

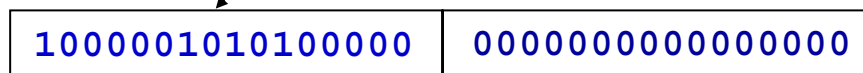
In SPIM wird die
Konvention des
unterliegenden
Rechners verwendet.



Laden von 32-Bit Operanden

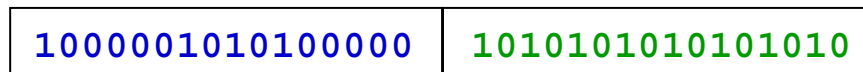
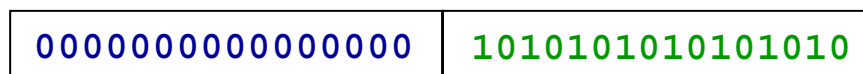
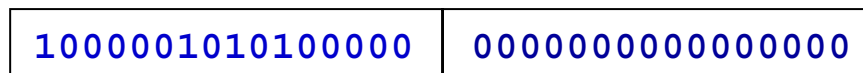
10000010101000001010101010101010 → \$t0

lui \$t0, 1000001010100000 (load upper immediate)



mit Nullen ausfüllen

ori \$t0, \$t0, 1010101010101010



Der globale Zeiger \$gp

Lade-Befehl (Pseudoinstruktion):

lw rt, address

lw \$v0, 0x10008000

Adresse liegt im Datensegment

Bisherige Lösung:

lui \$at, 0x1000

lui \$at, 0x1000

lw rt, Offset(\$at)

lw \$v0, 0x8000(\$at)

Offset := vier niedrigstwertige Stellen von adresse

Bessere Lösung:

lw rt, Offset(\$gp)

lw \$v0, 0(\$gp)

Der globale Zeiger \$gp enthält immer den Wert 1000 8000₁₆

→ Zugriff auf die Adressen 1000 0000₁₆ bis 1001 0000₁₆



Befehlssatz

Transportbefehle:

- 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`

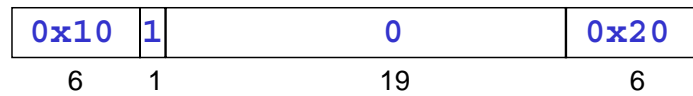


Befehlssatz

Unterbrechungs- und Ausnahmebehandlung:

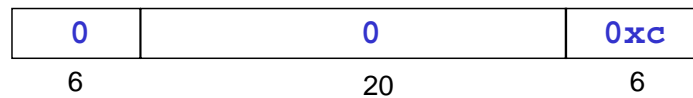
- Wiederherstellen des Status-Registers nach einer Unterbrechung

rfe



- Systemaufruf

syscall



- Software-Unterbrechung

break code



- Dummy-Operation

nop

