

Übung 3

□ MIPS-Assembler

- Befehlssatz
- Pseudobefehle
- Programmiertechniken
- Assemblerprogrammierung
- Stackprogrammierung
- Unterprogramm-Aufruf



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,rscr2`
- 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

| | |
|------------------------------------|-------------------------------------|
| <code>bgt rsrc1,rsrc2,label</code> | <code>bgtu rsrc1,rsrc2,label</code> |
| <code>blt rsrc1,rsrc2,label</code> | <code>bltu rsrc1,rsrc2,label</code> |

(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

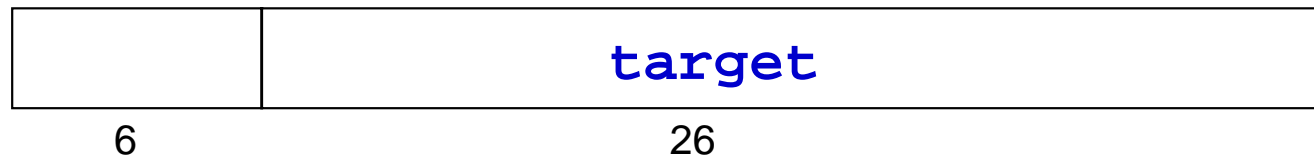
Branch



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, markel
{
    ....
} 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

| | |
|------------------------------|-----------------------------|
| <code>lb rt, address</code> | <code>sb rt, address</code> |
| <code>lbu rt, address</code> | |
| <code>lh rt, address</code> | <code>sh rt, address</code> |
| <code>lhu rt, address</code> | |
| <code>lw rt, address</code> | <code>sw rt, address</code> |
| <code>ld rt, address</code> | <code>sd rt, address</code> |
- Laden und Speichern von Koprozessor-Registern

| | | |
|-------------------------------|-------------------------------|------------|
| <code>lwcx rt, address</code> | <code>swcx rt, address</code> | (z=1, FPU) |
|-------------------------------|-------------------------------|------------|

Befehlssatz

Lade- und Speicherbefehle:

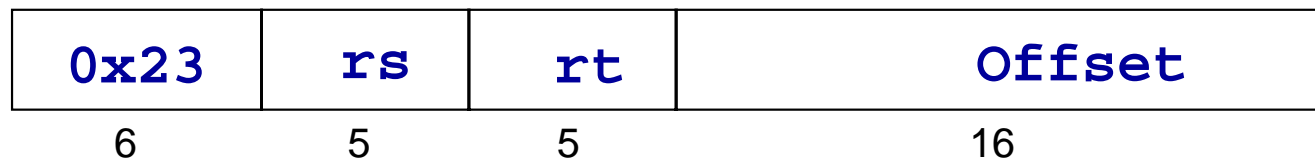
- Laden und Speichern von/an nicht-ausgerichtete Adressen
 - `lwl rt, address`
 - `lwr rt, address`
 - `ulh rdest, address` `ush rsrc, address`
 - `ulhu rdest, address`
 - `ulw rdest, address` `usw rsrc, address`
 - `ulhu rdest, address`

Lade-und Speicherbefehle

- Laden/Speichern von Bytes, Halbwörter und Wörter

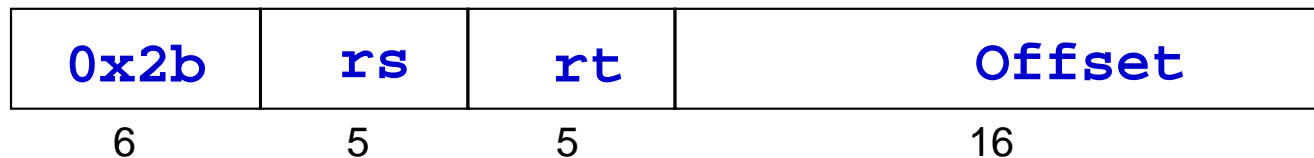
lw rt, address

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
```

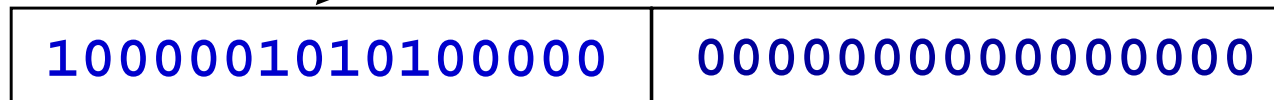
```
# $a3 enthaelt 0xef
```

In SPIM wird die
Konvention des
unterliegenden
Rechners verwendet.

Laden von 32-Bit Operanden

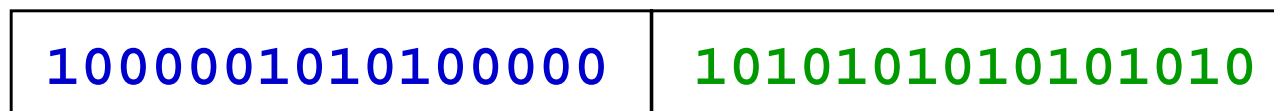
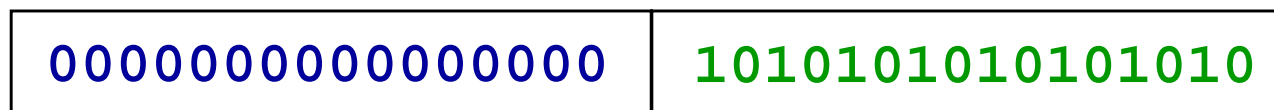
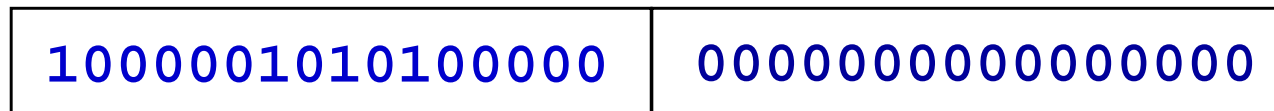
10000010101000001010101010101010 → \$t0

lui \$t0, 1000001010100000 (load upper immediate)



mit Nullen ausfüllen

ori \$t0, \$t0, 1010101010101010



Der globale Zeiger \$gp

Lade-Befehl (Pseudoinstruktion):

```
lw rt, address
```

Adresse liegt im Datensegment

```
lw $v0, 0x10008000
```

Bisherige Lösung:

```
lui $at, 0x1000
```

```
lw rt, Offset($at)
```

```
lui $at, 0x1000
```

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

Offset := vier niedrigstwertige Stellen von address

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`

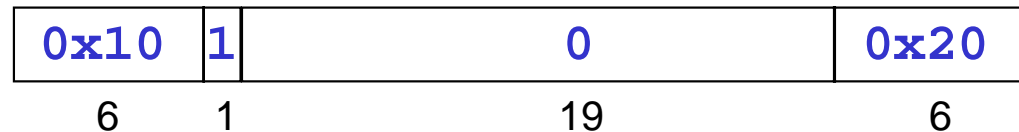
`s.s fdest, address`

Befehlssatz

Unterbrechungs- und Ausnahmebehandlung:

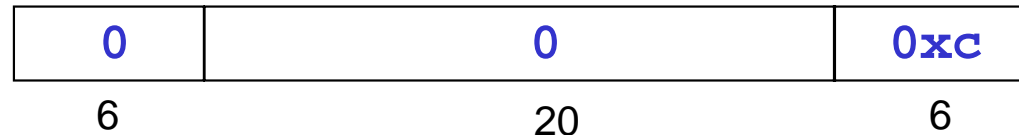
- Wiederherstellen des Status-Registers nach einer Unterbrechung

rfe



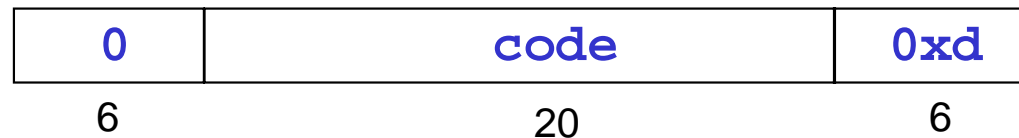
- Systemaufruf

syscall



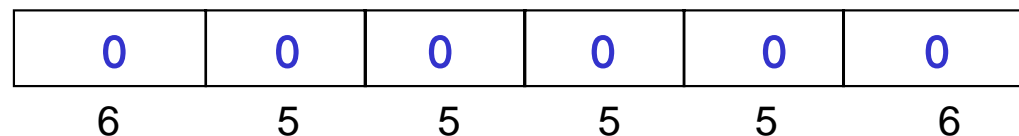
- Software-Unterbrechung

break code



- Dummy-Operation

nop



Wichtige MIPS-Befehle

□ Arithmetik:

- `add rd, rs, rt` bzw. `addi rt, rs, imm`
- `mul rdest, rsrc1, src2`
- `sub rd, rs, rt`

□ Verzweigungen

- `slt rd, rs, rt` bzw. `slti rd, rs, imm`
- `beq rs, rt, label`
- `bne rs, rt, label`

□ Sprünge

- `j target`
- `jal target`
- `jr rs`

□ Lade-Speicherbefehle (Transportbefehle)

- `lui rt, imm`
- `lw rt, address`
- `sw rt, address`



Ersetzung von Pseudoinstruktionen

| Pseudoinstruktion | wird ersetzt durch |
|--------------------------|--|
| <code>move Rd, Rs</code> | <code>addu Rd, \$0, Rs</code> |
| <code>neg Rd, Rs</code> | <code>sub Rd, \$0, Rs</code> |
| <code>b sym</code> | <code>bgez \$0, sym</code> |
| <code>li Rd, Imm</code> | <code>ori Rd, \$0, Imm</code> |
| <code>la Rd, sym</code> | <code>lui \$at, [sym / 10000₁₆] ori Rd, \$at, sym & FFFF₁₆</code> |
| <code>l.d Fd, sym</code> | <code>lui \$at, [sym / 10000₁₆] lwcl Fd, sym & FFFF₁₆(\$at) lui \$at, [sym / 10000₁₆] lwcl Fd + 1, sym & FFFF₁₆(\$at)</code> |

Ersetzung von Pseudoinstruktionen

| Pseudoinstruktion | wird ersetzt durch |
|-------------------------------|---|
| <code>bge Ra, Rb, sym</code> | <code>slt \$at, Ra, Rb</code> <code>beq \$at, \$0, sym</code> |
| <code>abs Rd, Rs</code> | <code>addu Rd, \$0, Rs</code> <code>bgez Rs, lbl</code> <code>sub Rd, \$0, Rs</code> <code>lbl:</code> |
| <code>rol Rd, Rs, dist</code> | <code>srl \$at, Rs, 32 - dist</code> <code>sll \$at, Rd, Rs, dist</code> <code>or Rd, Rd, \$at</code> |
| <code>rem Rd, Ra, Rb</code> | <code>bne Rb, \$0, lbl</code> <code>break 0</code> <code>lbl: div Ra, Rb</code> <code>mfhi Rd</code> |
| <code>nop</code> | <code>or \$0, \$0, \$0</code> |

Programmiertechniken

C:

```
if( ) {...}
  else { if( ) {...}
        else { if( ) {...}
              else {...}
            }
        }
    }
```

C:

```
switch (note)
{
    case1:    1-Anweisungen
              break;
    case2:    2-Anweisungen
              break;
    ...
    ...
    ...
    case6:    6-Anweisungen
              break;
    default:  Fehlermeldung
              break;
}
```



Programmiertechniken

```
# Note steht in der
# Variablen note

lw    $t0, note
li    $t1, 1
li    $t2, 2
      ....
li    $t5, 5

beq   $t0, $t1, marke1
beq   $t0, $t2, marke2
      ....
      ....
      ....

beq   $t0, $t5, marke5
      .... # default

b     weiter
```

```
marke1:  ....      # Note 1
          ....
          b weiter

marke2:  ....      # Note 2
          ....
          b weiter

          .
          .
          .
          .

marke5:  ....      # Note 5
          ....
          b weiter

weiter:  ....      # Mat.-Nr. ....
          ....      # hat die Note:
```

Programmiertechniken

c-Code

```
int x = 12;  
int y = 34;  
x = x + y;
```

MIPS-Assembler

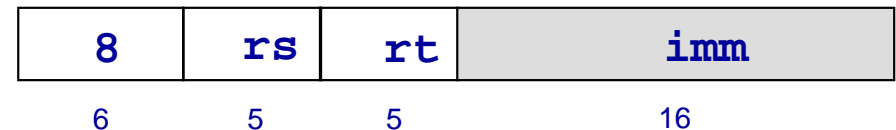
```
addi $t1, $zero, 12  
addi $t2, $zero, 34  
add  $t1, $t1, $t2
```

Programmiertechniken

Register \$t1 und \$t2 in den Speicher ab
der Adresse 0x1000 0004

```
addi $s1, $zero, 0x1000 0004
```

addi rt,rs,imm



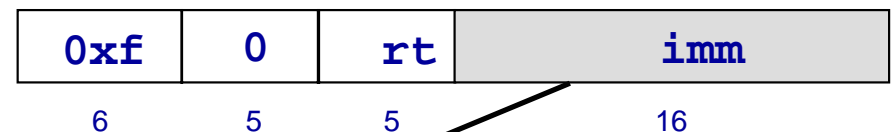
```
lui $s1, 0x1000
```

```
ori $s1, 0x0004
```

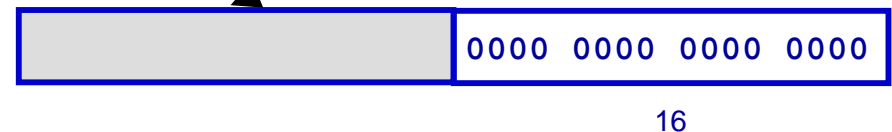
```
sw $t1, 0($s1)
```

```
sw $t2, 4($s1)
```

lui rt,rs,imm



\$s1



Programmiertechniken

c-Code

```
if (a < b)
    x = b;
else
    x = a;
```

MIPS-Code

```
        lw $t0, a           # initialisiere $t0
        lw $t1, b           # initialisiere $t1
        slt $t2, $t0, $t1   # ($t0 < $t1)?
        beq $t2, $zero, else # wenn die Bedingung nicht erfuehlt
                             # ist, gehe zu else
        add $t3, $t1, $zero  # es gilt: ($t0 < $t1)
        j cont
else:    add $t3, $t0, $zero  # es gilt ($t0 >= $t1)
                             # move $t0 to $t3 (result)

cont:    ...
```

Programmiertechniken

C- Code

```
x[0] = x[1] + x[2];
```

MIPS Assemblercode:

- Annahme: Startadresse n steht bereits in \$t1

```
lw $t2, 4($t1)
```

```
lw $t3, 8($t1)
```

```
add $t4, $t2, $t3
```

```
sw $t4, 0($t1)
```

