3. Übung

- MIPS- Programmstruktur
- Befehlssatz
  - Literatur: Hennessy & Patterson (Anhang A auf der TI-Homepage)

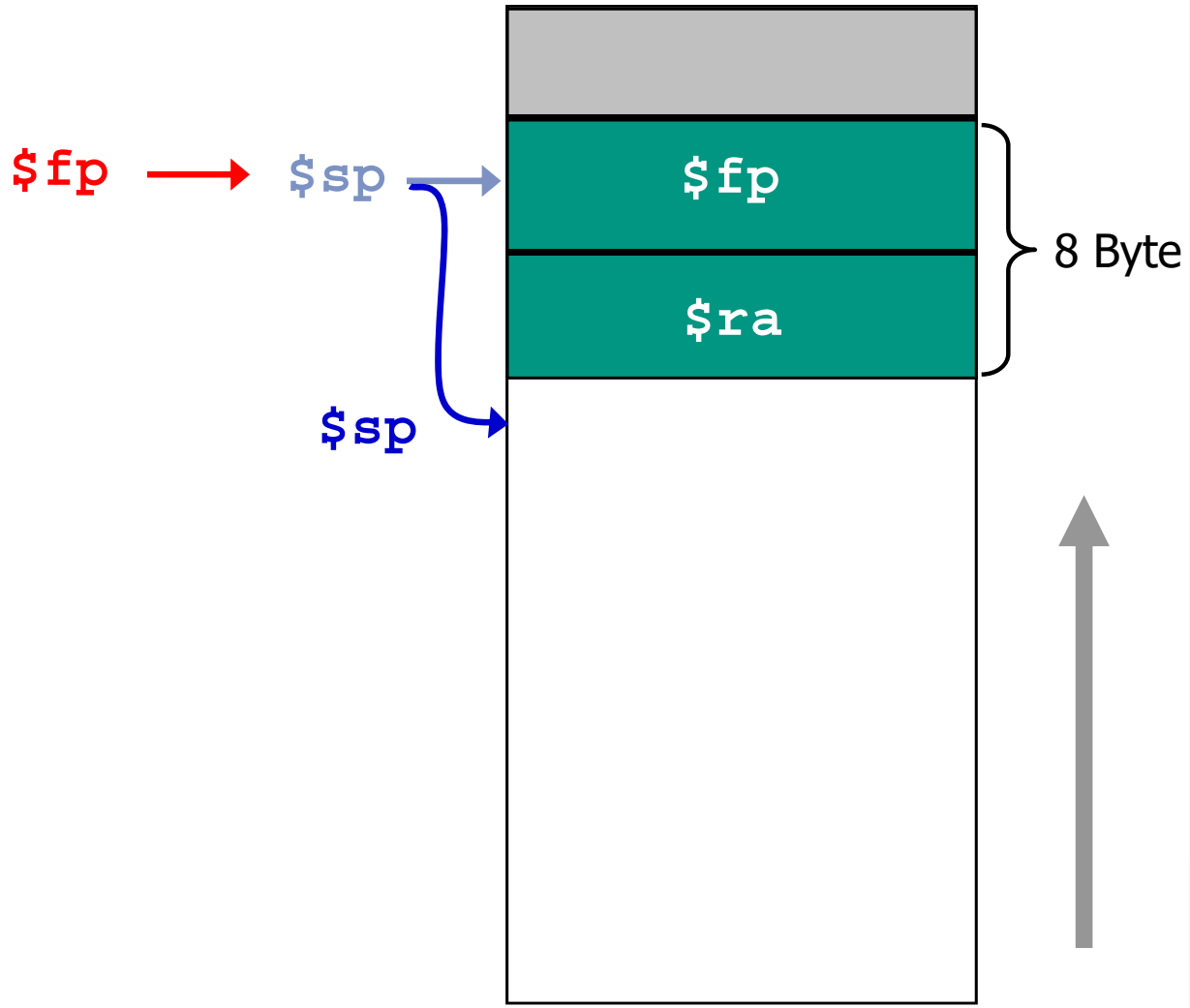
- Programmiertechniken in Assembler
- Stack-Programmierung
- Unterprogramme in MIPS
- Ausnahme- und Unterbrechungsbehandlung

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
lf_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 div 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, lf_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       # $a0 = $s0 + $s1
jal print_int

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

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

beqz $s1, error
la $a0, result_div      # Ausgabe A mod B
jal print_str
div $s0, $s1            # LO = $s0 div $s1   (Rest in HI)
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
```

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, rscr2
```

- Vergleich eines Registers mit Null

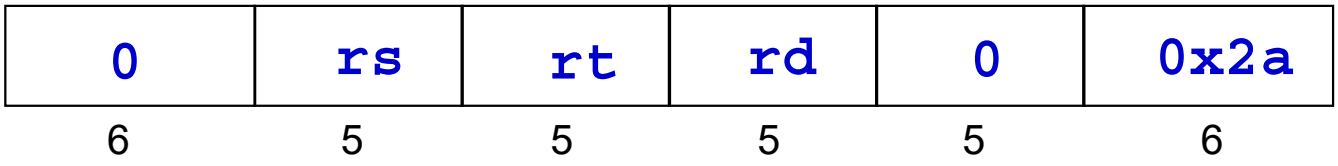
Befehlssatz

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

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

`slt rd, rs, rt`

slt: set less than



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 z wahr/falsch ist

```
bczt label        bczf label
```

bc1t 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`)

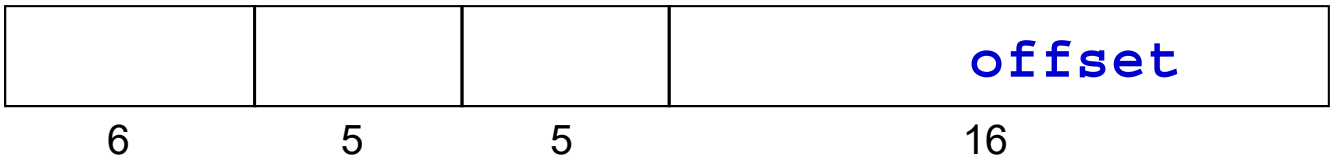
Befehlssatz

Kontrollflussbefehle

- Verzweigen, wenn zwei Register gleich/ungleich sind
  `beq rs,rt,label` `bne 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-bit, vorzeichenbehaftet

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

jump

26-bit Adress-Feld

Befehlssatz

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
```

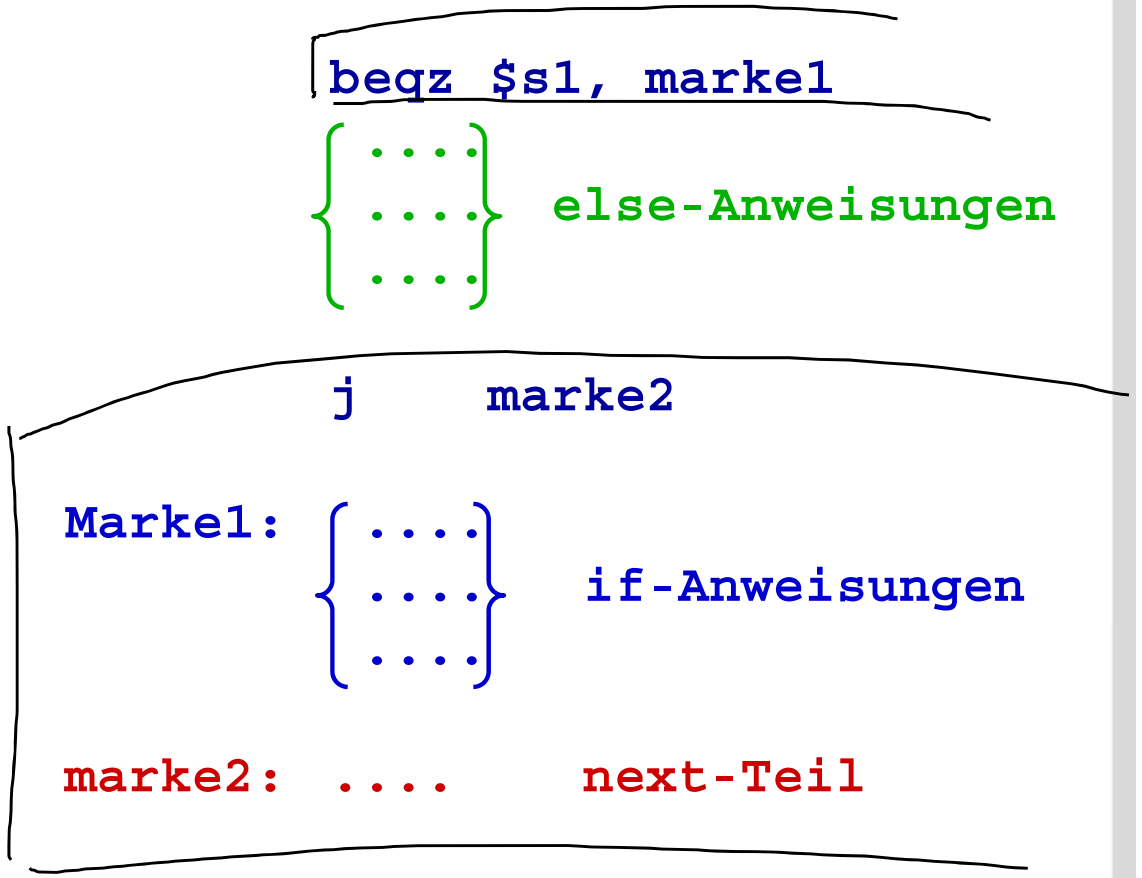
Befehlssatz

Zweistufige Verzweigung:

if-else-Anweisung

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

next;
```

Assembler-Code



Befehlssatz

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:       ....
```

Befehlssatz

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

```
lwcz rt, address    swcz rt, address     (z=1, FPU)
```

lwc1

Befehlssatz

Lade- und Speicherbefehle:

- Laden und Speichern von/an nicht-ausgerichtete Adressen

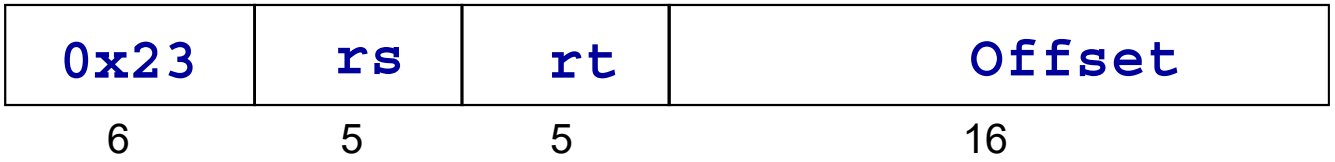
```
lwl rt, address
lwr rt, address
ulh rdest, address        ush rsrc, address
ulhu rdest, address
ulw rdest, address        usw rsrc, address
```

Lade-und Speicherbefehle

- Laden/Speichern von Bytes, Halbwörter und Wörter

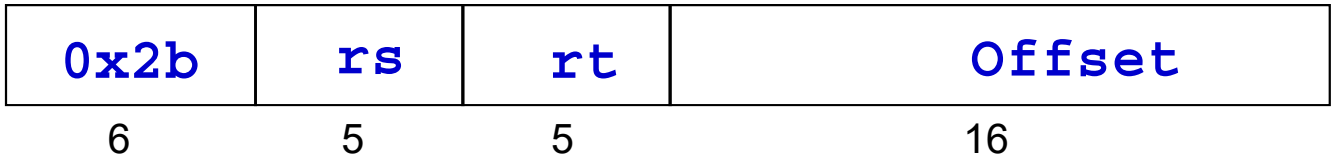
```
lw rt, address
```

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 :

A[0] → ($s3)

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

MIPS-Code:

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

32 = 8 · 4

Unterschied zwischen lb und lbu

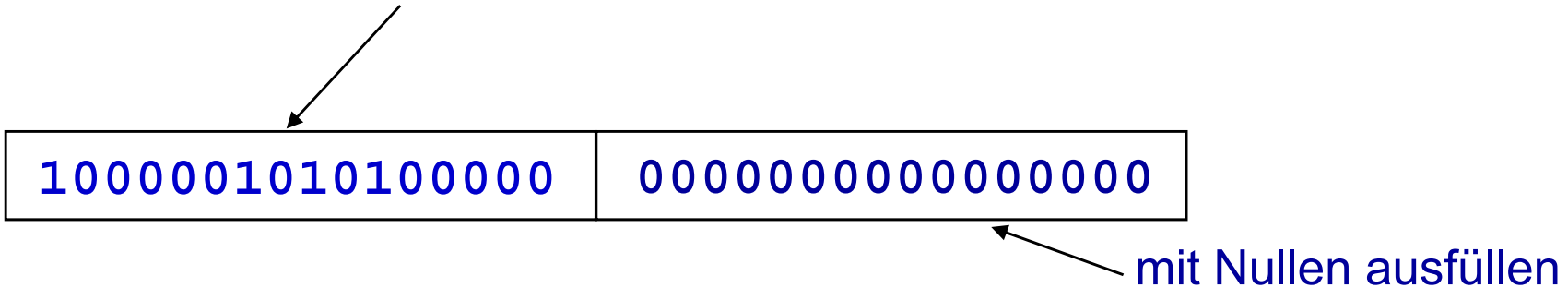
```
            .data
result:     .word 0x89abcdef    0x79abcdef
            .text

# Start des Hauptprogrammes

            .globl main
main:       ...
            lbu $a0, result
            # $a0 enthaelt 0x00000089    0x00000079
            ...
            lb $a0, result
            # $a0 enthaelt 0xffffff89    0x00000079
            ...
            jr $ra
```

Laden von 32-Bit-Operanden

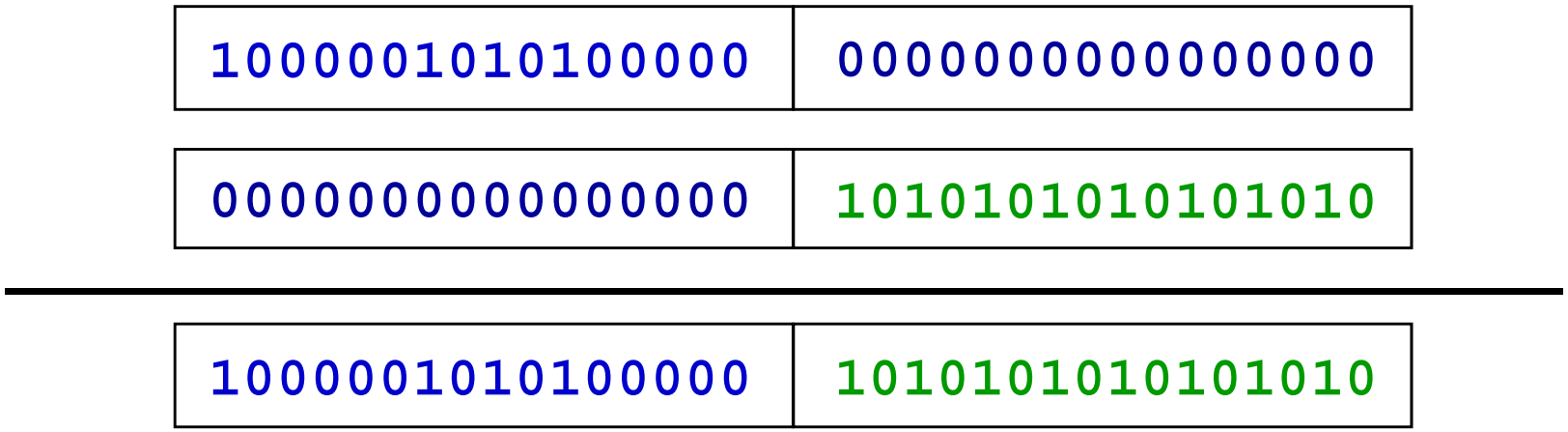
```
10000010101000001010101010101010 → $t0
```

```
lui $t0, 1000001010100000
```

(load upper immediate)




```
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 `addresse`

Bessere Lösung:

```
lw rt, Offset($gp)              lw $v0, 0($gp)
```

Der globale Zeiger $gp enthält immer den Wert $1000\ 8000_{16}$

➔ Zugriff auf die Adressen $1000\ 0000_{16}$ bis $1001\ 0000_{16}$

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 Koprozessor z

```
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

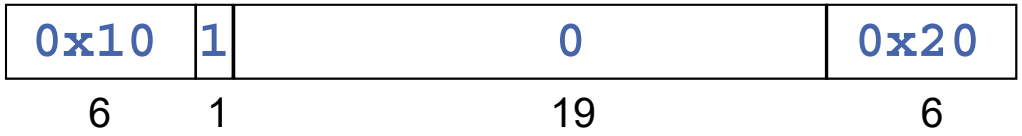
```
l.d fdest, address   l.s fdest, address
s.d fdest, address   s.s fdest, address
```

Befehlssatz

Unterbrechungs- und Ausnahmebehandlung:

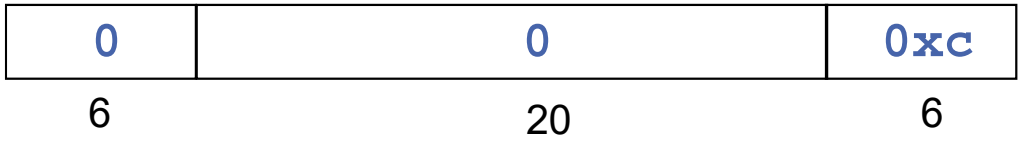
- Wiederherstellen des Status-Registers nach einer Unterbrechung

  `rfe`



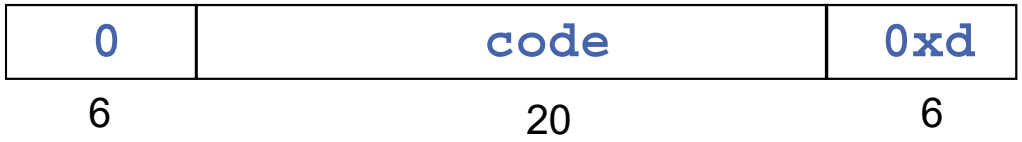
- Systemaufruf

  `syscall`



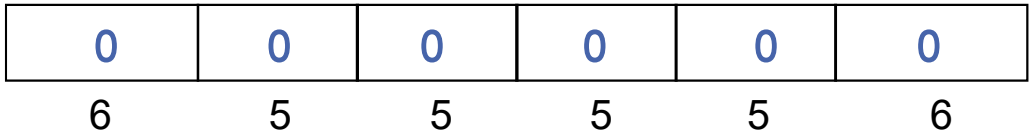
- Software-Unterbrechung

  `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
`move Rd, Rs`	`addu Rd, $0, Rs`
`neg Rd, Rs`	`sub Rd, $0, Rs`
`b sym`	`bgez $0, sym`
`li Rd, Imm`	`ori Rd, $0, Imm`
`la Rd, sym`	`lui $at, [sym / 10000`$_{16}$`]` `ori Rd, $at, sym & FFFF`$_{16}$

`l.d Fd, sym`	`lui $at, [sym / 10000`$_{16}$`]`
	`lwc1 Fd, sym & FFFF`$_{16}$`($at)`
	`lui $at, [sym / 10000`$_{16}$`]`
	`lwc1 Fd + 1, sym & FFFF`$_{16}$`($at)`

Ersetzung von Pseudoinstruktionen

Pseudoinstruktion	Wird ersetzt durch
`bge Ra, Rb, sym`	`slt $at, Ra, Rb` `beq $at, $0, sym`

`abs Rd, Rs`	`addu Rd, $0, Rs`
	`bgez Rs, lbl`
	`sub Rd, $0, Rs`
	`lbl:`

`rol Rd, Rs, dist`	`srl $at, Rs, 32 - dist`
	`sll $at, Rd, Rs, dist`
	`or Rd, Rd, $at`

`rem Rd, Ra, Rb`	`bne Rb, $0, lbl`
	`break 0`
	`lbl: div Ra, Rb`
	`mfhi Rd`

`nop`	`or $0, $0, $0`

3. Übung – 2. Teil

Assemblerprogrammierung mit dem MIPS-Simulator MARS

- Programmiertechniken in Assembler
- Stack-Programmierung
- Unterprogramme in MIPS
- Ausnahme- und Unterbrechungsbehandlung

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

Karlsruher Institut für Technologie

C-Code:

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

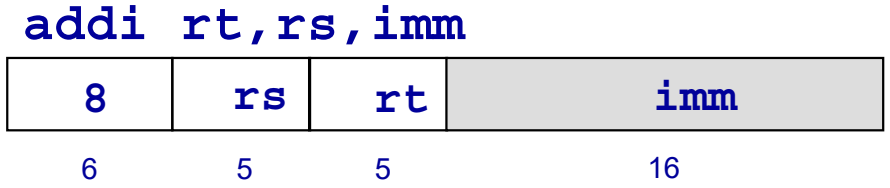
MIPS-Assembler:

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

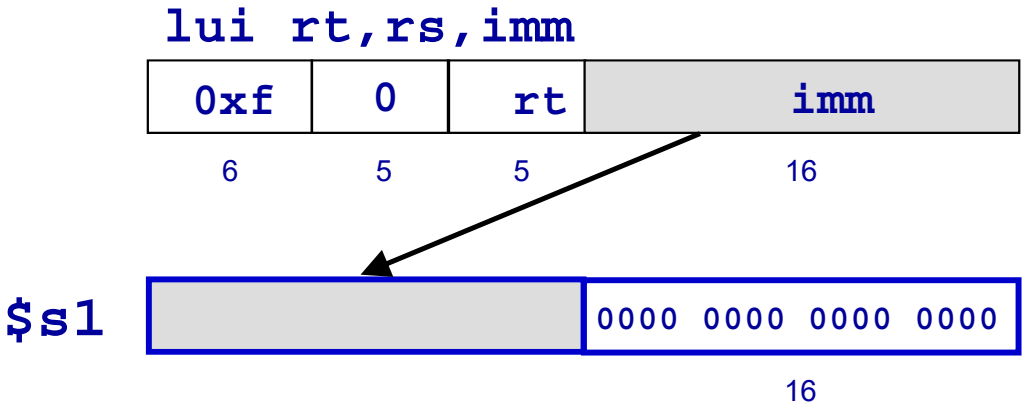
Programmiertechniken

Register $t1 und $t2 im Speicher ab der Adresse 0x1000 0004

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




```
lui $s1, 0x1000
ori $s1, 0x0004
sw $t1, 0($s1)
sw $t2, 4($s1)
```



Programmiertechniken

C-Code:

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

MIPS-Assembler:

```
        lw $t0, a              # initialisiere $t0
        lw $t1, b              # initialisiere $t1
        slt $t2, $t0, $t1      # ($t0 < $t1)?
        beq $t2, $zero, else   # wenn die Bedingung nicht erfuellt
                               # 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:   …
```

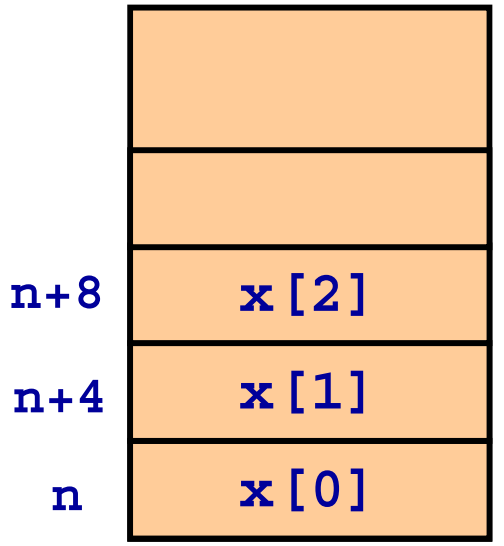
C-Code:

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

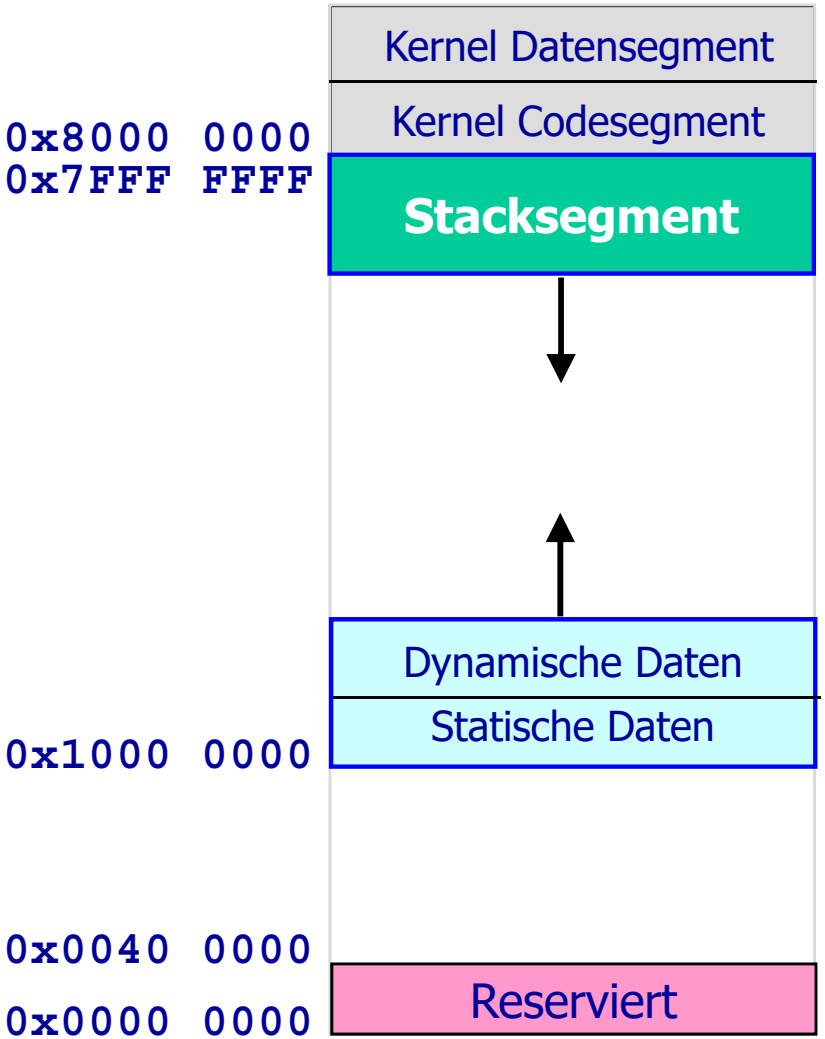
MIPS-Assembler:

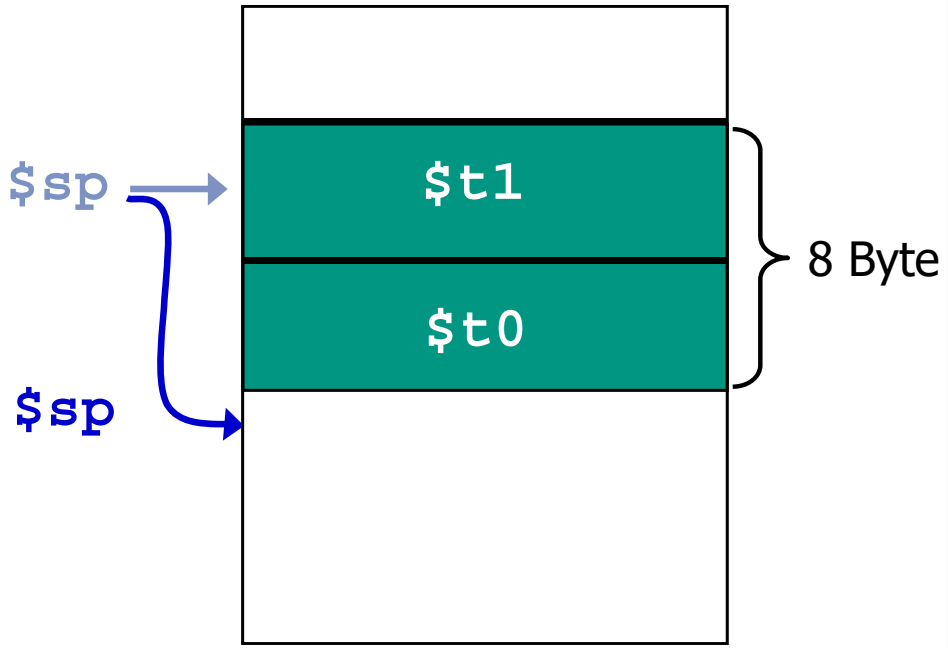
- Annahme: Startadresse n steht bereits in $t1

```
lw $t2, 4($t1)
lw $t3, 8($t1)
add $t4, $t2, $t3
sw $t4, 0($t1)
```



Stackprogrammierung

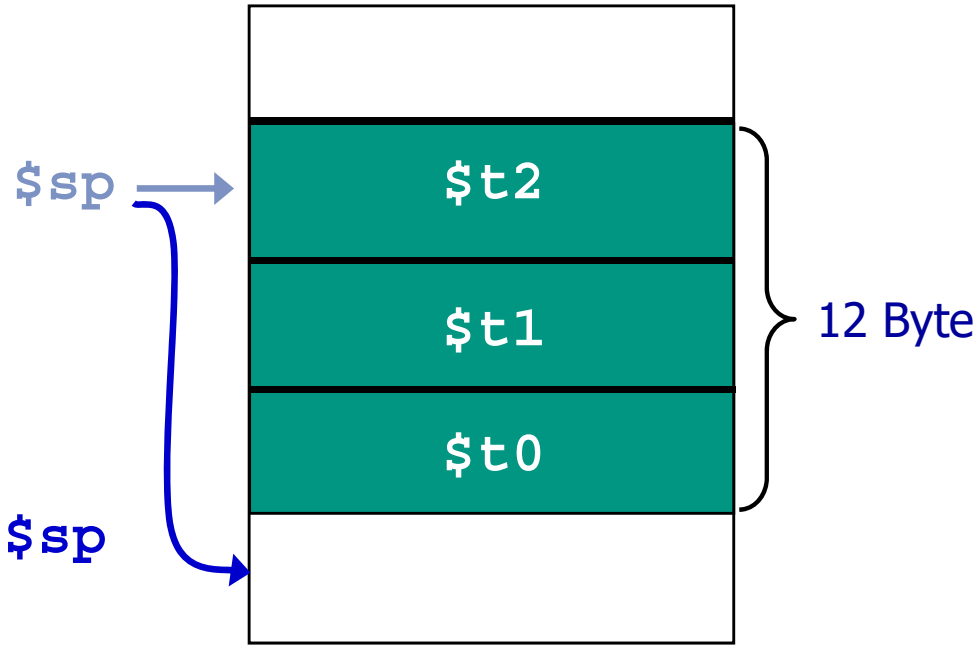


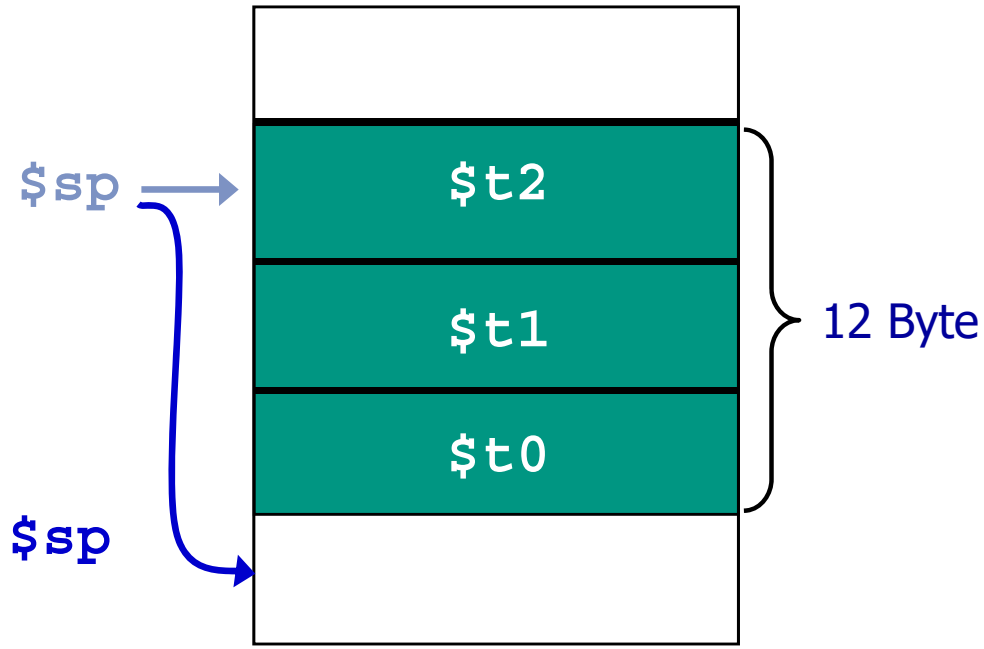



```
subu   $sp, $sp, 8
sw     $t0, 4($sp)
sw     $t1, 8($sp)
```

Stackprogrammierung

- **Lesen vom Stack:**






```
subu   $sp, $sp, 12
sw     $t0, 4($sp)
sw     $t1, 8($sp)
sw     $t2, 12($sp)
```

```
lw     $t0, 4($sp)
lw     $t1, 8($sp)
lw     $t2, 12($sp)
addi   $sp, $sp, 12
```

Unterprogrammaufrufe

MIPS-Assembler:

```
        j min       # Sprung zu min
        ...         # Befehl nach Berechnung von min

min:    ...         # Code fuer min(a,b)
        ...
        j ??        # Zurueck zum Hauptprogramm
```

Probleme:

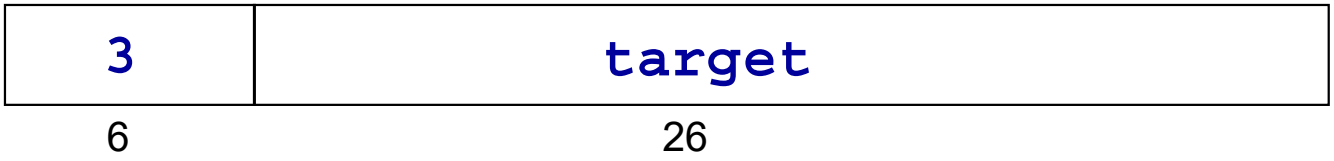
- Rücksprungadresse sichern und wiederherstellen
- Parameterübergabe

Unterprogrammaufrufe

- **Rücksprungadresse:** MIPS unterstützt den Funktionsaufruf durch speziellen Befehl

```
jal ("jump and link")
```

```
jal target
```



Springe zu target und speichere die Adresse des nächsten Befehls in $ra

Unterprogrammaufrufe

```
main:       ...
            jal sub
            ...

            jal sub
            ...

sub:        ...
```

```
            jr $ra
```

Unterprogrammaufrufe

- Regeln oder Konventionen zur Verwendung von **Registern** und **Kellerspeicher (Stack)** bei einem Unterprogrammaufruf
- Informationen bezüglich des Unterprogramms und des unterbrochenen Programms werden in einem **Rahmen (Frame)** auf dem Stack gespeichert. Dies sind z.B.:
  - Argumente des Unterprogramms
  - Lokale Variablen des Unterprogramms
  - Registerinhalte, die vom Unterprogramm nicht verändert werden dürfen

Unterprogrammaufrufe

- **Parameterübergabe**:

  Konventionen zur Verwendung der Register, um Fehler zu vermeiden

  - $a0 - $a3: Register für Argumente des Unterprogramms
  - $v0, $v1: Register für Funktionswert bzw. für Ausgabeparameter des Unterprogramms
  - $t0 - $t9: Register können im Unterprogramm überschrieben werden
  - $s0 - $s7: Register dürfen nicht überschrieben werden (bzw. müssen gesichert/wiederhergestellt werden)

Einfacher Unterprogrammaufruf

```
main:       jal subroutine


subroutine: # keine weiteren
            # Unterprogrammaufrufe

            # für lokale Variablen
            # nur $t0 bis $t9

            jr $ra
```

Unterprogrammaufrufe

Beispiel (C-Code):

```
int min(int a, int b)
{
  if (a < b) return a;
     else return b;
}

main()
{
  int x = 123;
  int y = 456;
  int z = min(x, y);
}
```

Unterprogrammaufrufe

```
lw $s0, x                    # initialisiere $s0 mit x=123
lw $s1, y                    # initialisiere $s1 mit y=456
add $a0, $s0, $zero          # kopiere $s0 in Register $a0
                             # fuer Parameteruebergabe
add $a1, $s1, $zero          # entsprechend $s1 in $a1
jal min                      # springe zur Funktion min
add $s2, $v0, $zero          # kopiere Funktionswert in
                                  # Ergebnisregister
sw $s2, z                    # speichere Ergebnis ab
...
```

Unterprogrammaufrufe

```
min:    add $t0, $zero, $a0          # initialisiere $t0 mit
                                     # Parameter a
        add $t1, $zero, $a1          # initialisiere $t1 mit
                                     # Parameter b
        slt $t2, $t1, $t0            # ($t1 < $t0)?
        beq $t2, $zero, else                 # wenn die Bedingung
   nicht                                             # erfuellt ist,
   gehe zu else
        add $v0, $t1, $zero          # es gilt: ($t1 < $t0)
        j ende                               #

else:   add $v0, $t0, $zero          # es gilt ($t1 >= $t0)
                                     # move $t0 to $v0 (result)
ende:   j $ra                        # Ruecksprung
```

Unterprogrammaufrufe

Ein Argument ist eine Integer- oder Fließkomma-Zahl oder ein Zeiger auf eine größere Datenstruktur (z. B. Zeichenkette)

Aufgaben des Aufrufers:

- *Temporäre* Register `$t0` bis `$t9` sichern, falls gültige Daten darin enthalten sind
- Übergabe der ersten vier Argumente in den Registern `$a0` bis `$a3` (ggf. als Zeiger auf das Argument)
- Register `$a0` bis `$a3` sichern, falls die Argumente später noch gebraucht werden
- Übergabe weiterer Argumente auf dem Stack

Unterprogrammaufrufe

Aufruf eines Unterprogramms an der Adresse <addr>
mit dem Befehl

```
jal <addr>
```

Die Adresse des nachfolgenden Befehls wird im Register
`$ra` gespeichert.

Informationen bezüglich des Unterprogramms werden in einem Rahmen auf dem Stapel gespeichert.

Rahmengröße := (Anzahl der Argumente +
Anzahl der zu sichernden Register +
Anzahl der lokalen Variablen) x 4

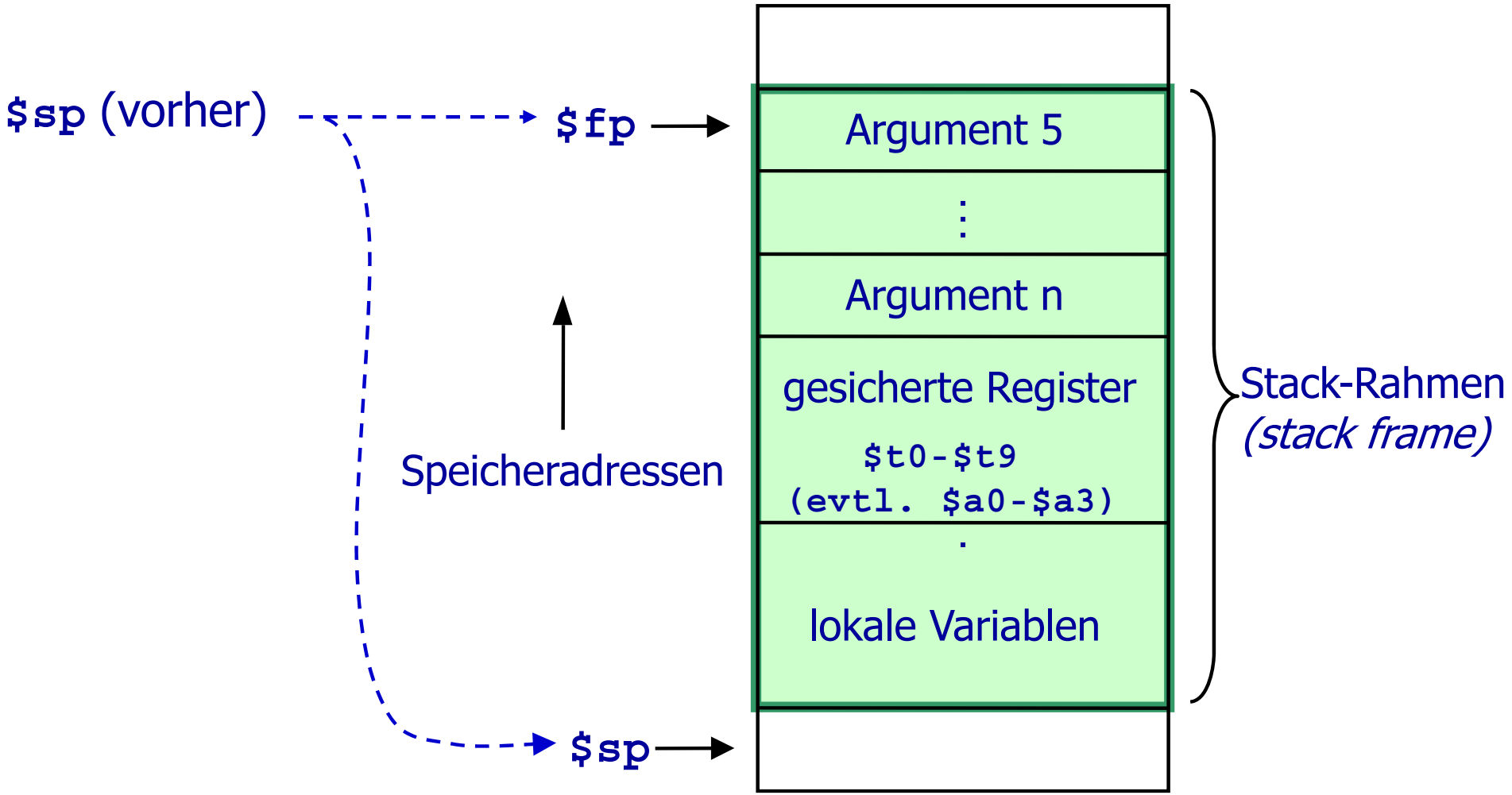
(Zu sichernde Fließkommazahlen mit doppelter Genauigkeit benötigen acht Bytes Speicherplatz)

Unterprogrammaufrufe

Aufgaben des Aufgerufenen:

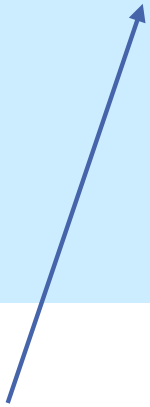
- Reservieren von Speicherplatz für einen neuen Rahmen durch Subtraktion der Rahmengröße von Stack-Pointer (`$sp`)
- Sichern der Register `$s0` bis `$s7`, falls diese im Unterprogramm verwendet werden
- Sichern des Rücksprungadressenregisters `$ra`, falls das Unterprogramm weitere Unterprogramme aufruft
- Sichern des Rahmenzeigers `$fp`
- Anlegen eines neuen Rahmenzeigers durch Addition der Rahmengröße zum Stapelzeiger

Stack-Rahmen (*stack frame*)



Beispiel für einen Unterprogrammaufruf

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

            jal subroutine

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

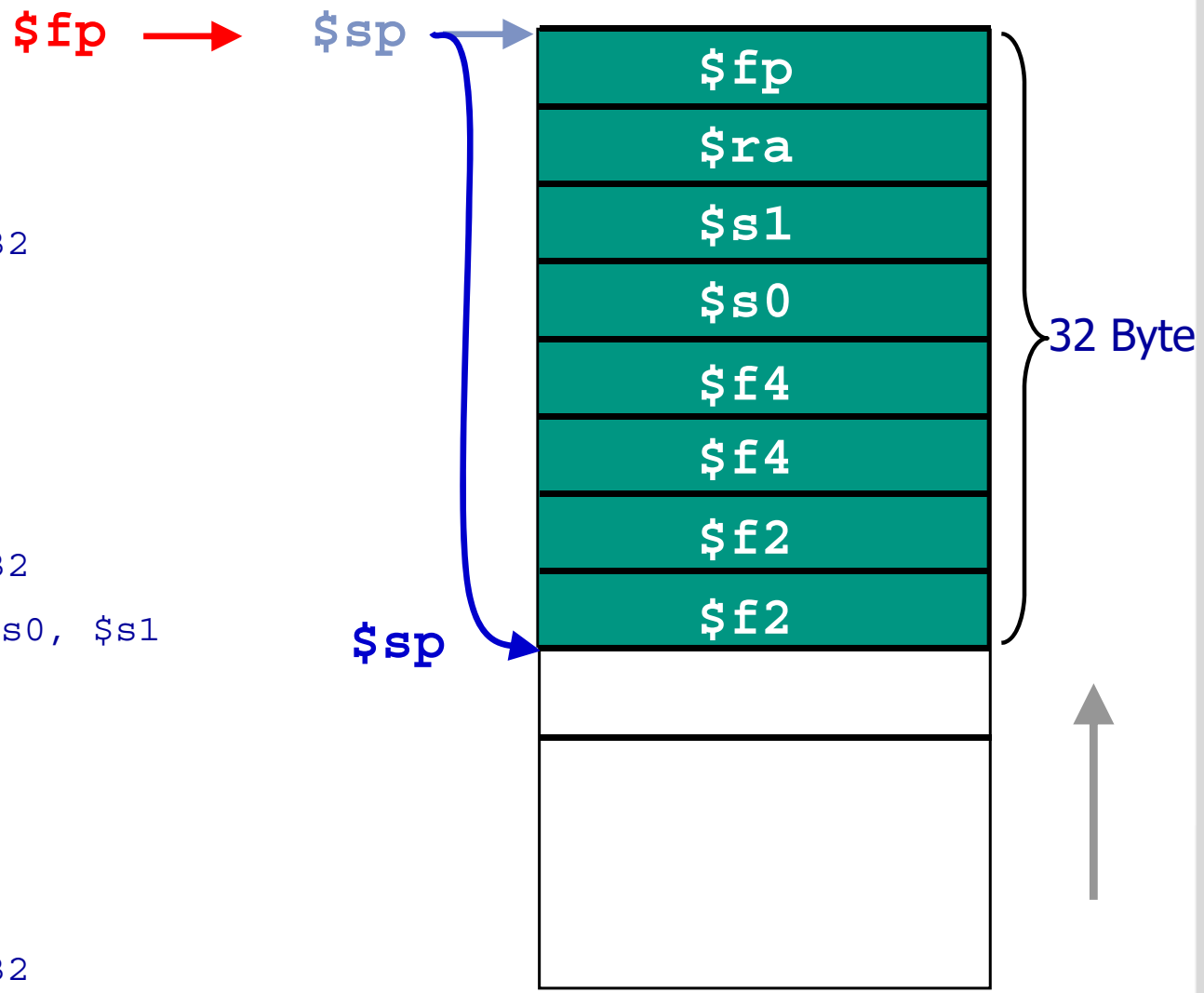


Unterprogrammaufrufe mit jal-Befehl!

```
                .text
subroutine:  subu $sp, $sp, 32
             s.d $f2, 4($sp)
             s.d $f4, 12($sp)
             sw $s0, 20($sp)
             sw $s1, 24($sp)
             sw $ra, 28($sp)
             sw $fp, 32($sp)
             addu $fp, $sp, 32

# Veraenderung von $f2,$f4,$s0,$s1

             l.d $f2, 4($sp)
             l.d $f4, 12($sp)
             lw $s0, 20($sp)
             lw $s1, 24($sp)
             lw $ra, 28($sp)
             lw $fp, 32($sp)
             addu $sp, $sp, 32
             jr $ra
```

Beispiel für einen Unterprogrammaufruf

```
                .text
subroutine:     subu $sp, $sp, 32
                s.d $f2, 4($sp)
                s.d $f4, 12($sp)
                sw $s0, 20($sp)
                sw $s1, 24($sp)
                sw $ra, 28($sp)
                sw $fp, 32($sp)
                addu $fp, $sp, 32
# Veraenderung von FP2, FP4, $s0, $s1
                l.d $f2, 4($sp)
                l.d $f4, 12($sp)
                lw $s0, 20($sp)
                lw $s1, 24($sp)
                lw $ra, 28($sp)
                lw $fp, 32($sp)
                addu $sp, $sp, 32
                jr $ra
```



Rekursive Unterprogrammaufrufe

- **Problem:**
  - Rücksprungadresse im Register `$ra` wird bei wiederholtem Unterprogrammaufruf immer wieder überschrieben
  - Entsprechendes gilt für lokale Daten in Registern
- **Lösung:**
  - Sichern von Rücksprungadressen und lokalen Daten auf einem Stack

Rekursive Unterprogrammaufrufe

C-Code:

```
main ()
{
  printf („Die Fakultaet von 10 ist: %d\n“, fakultaet(10));
}


int fakultaet (int n)
{
  if (n < 1)
    return (1);
  else
    return (n * fakultaet(n-1));
}
```

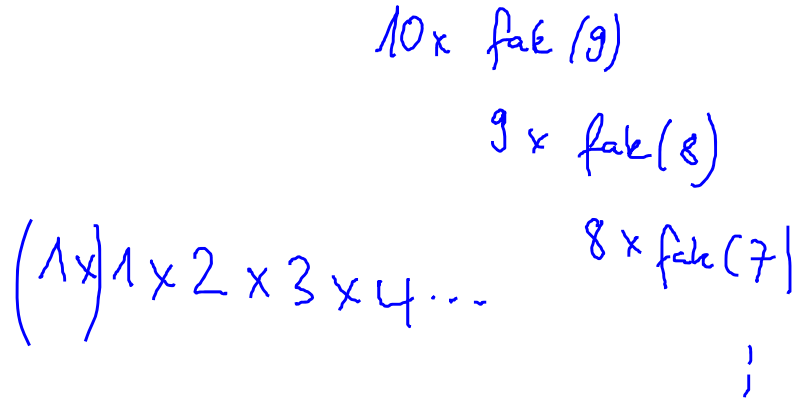
Rekursive Unterprogrammaufrufe

MIPS-Assembler:

```
fakultaet: subu $sp, $sp, 8      # Stack-Frame von 8 Bytes
           sw $ra, 8($sp)        # Ruecksprungadresse
           sw $a0, 4($sp)        # Argument(n) sichern

           slt $t0, $a0, 1       # (n < 1)? lade das Argument
           beqz $t0, L1          # falls n >=1 gilt, gehe zu L1

           add $v0, $zero, 1     # return 1

           lw $ra, 8($sp)        # Ruecksprungadresse wiederherstellen
           addu $sp, $sp, 8      # Frame-Stack loeschen
           jr $ra                # gehe zur Ruecksprungadresse
```

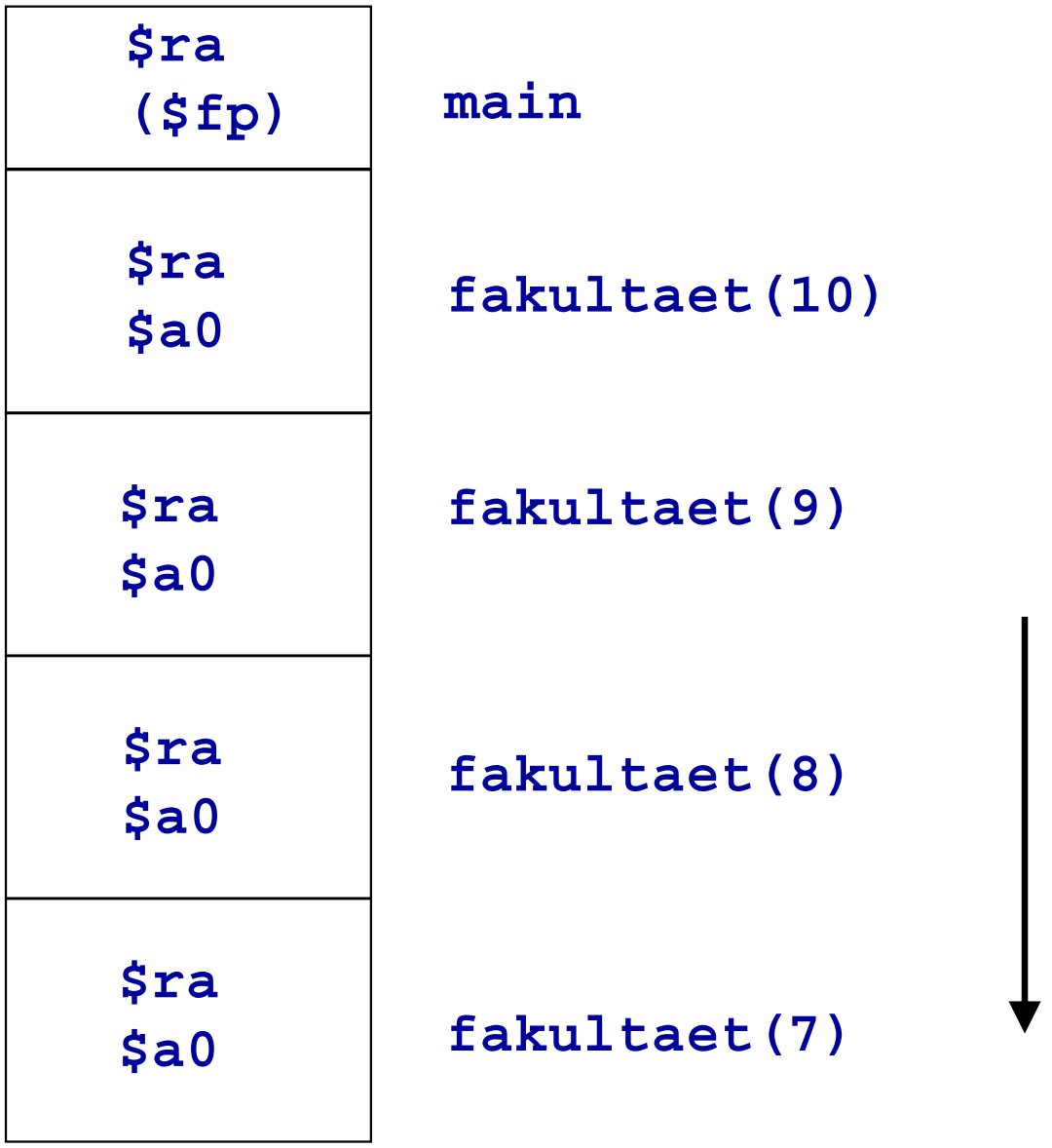
Rekursiver Unterprogrammaufruf

```
L1:     sub $a0, $a0, 1         # verwende n-1 als Argument
        jal fakultaet           # rufe mit n-1 auf

        lw $a0, 4($sp)          # hole Argument n wieder vom Stack
        lw $ra, 8($sp)          # hole Ruecksprungadresse vom Stack
        add $sp, $sp, 8         # Stack-Frame loeschen

        mul $v0, $a0, $v0       # return n*fak(n-1)

        jr $ra                  # Ruecksprung
```

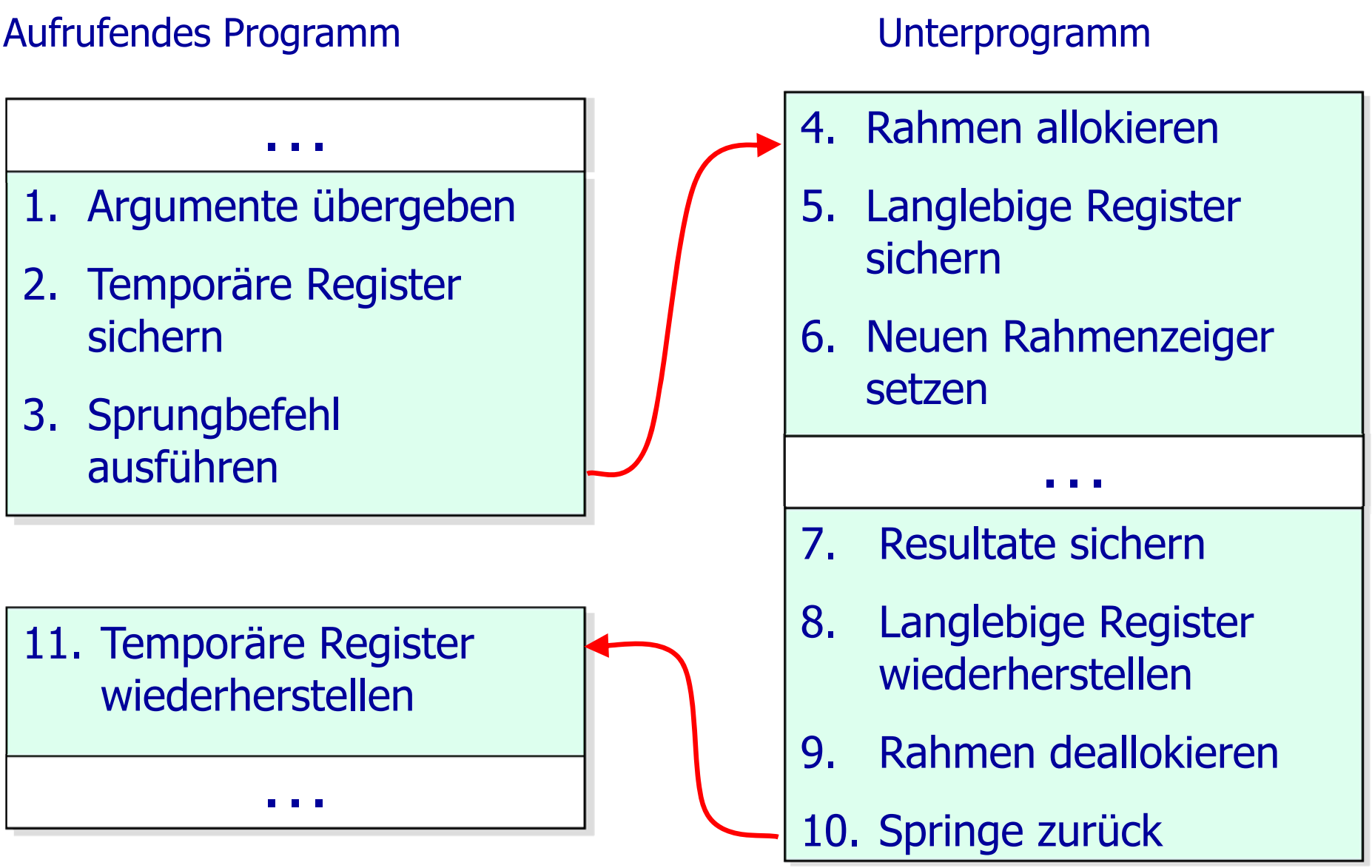


Beispiel 2

Stack



Unterprogrammaufruf



Ausnahme- und Unterbrechungsbehandlung

Karlsruher Institut für Technologie

Ausnahme (Exception, Trap):

- Jede unerwartete Änderung im Kontrollfluss
- Synchron zum Programmablauf, d.h. im Zusammenhang mit bestimmten Befehlen (z.B. syscall)
- **Beispiel:** ungültiger Befehl; arithmetische Fehler, …

Unterbrechung (Interrupt):

- Ausnahme, die durch einen äußeren Einfluss verursacht wurde
- Asynchron zum Programmablauf, d.h. steht nicht in direkter Abhängigkeit zu bestimmten Befehlen
- Behandlung durch das Betriebssystem
- **Beispiel:** Ein- und Ausgabegeräte (Drücken einer Taste)

MIPS - Ausnahmen

Code	Beschreibung
0	externe Unterbrechung
4	fehlerhafte Adresse beim Laden oder Befehl-Holen
5	fehlerhafte Adresse beim Speichern
6	Busfehler beim Befehl-Holen
7	Busfehler beim Laden oder Speichern
8	Systemaufruf
9	Programmunterbrechung (breakpoint) zur Fehlersuche
10	reservierter Befehl
12	arithmetischer Überlauf bei Ganzzahlberechnungen
14	ungültiges Fließkomma-Ergebnis
15	Division durch Null
16	Überlauf bei Fließkomma-Berechnung
17	Unterlauf bei Fließkomma-Berechnung

Software

Beispiele für Ausnahmen

Externe Unterbrechung (Code 0):

- Drücken einer Taste
- Maus wurde bewegt
- Timer ist abgelaufen
- Peripheriegerät (z. B. Drucker) ist fertig
- Serielle Schnittstelle hat ein Wort empfangen

Fehlerhafte Adresse (Code 4 & 5):

```
            .data
string:     .asciiz "arglwuz = "
result:     .space 23    # Not aligned!

            .text
            .globl main
main:       ...
            lw $a0, result
```

Beispiele für Ausnahmen

Busfehler (Code 6 & 7):

- Time-Out
- Paritätsfehler
- ungültige Port- oder Speicher-Adressen

Systemaufruf (Code 8):

Software-Interrupt zur Ausführung von Unterprogrammen des Betriebssystems

Programmunterbrechung (breakpoint) (Code 9):

Ausgabe von Register- oder Speicherinhalten oder des Prozessorstatus zur Fehlersuche

Beispiele für Ausnahmen

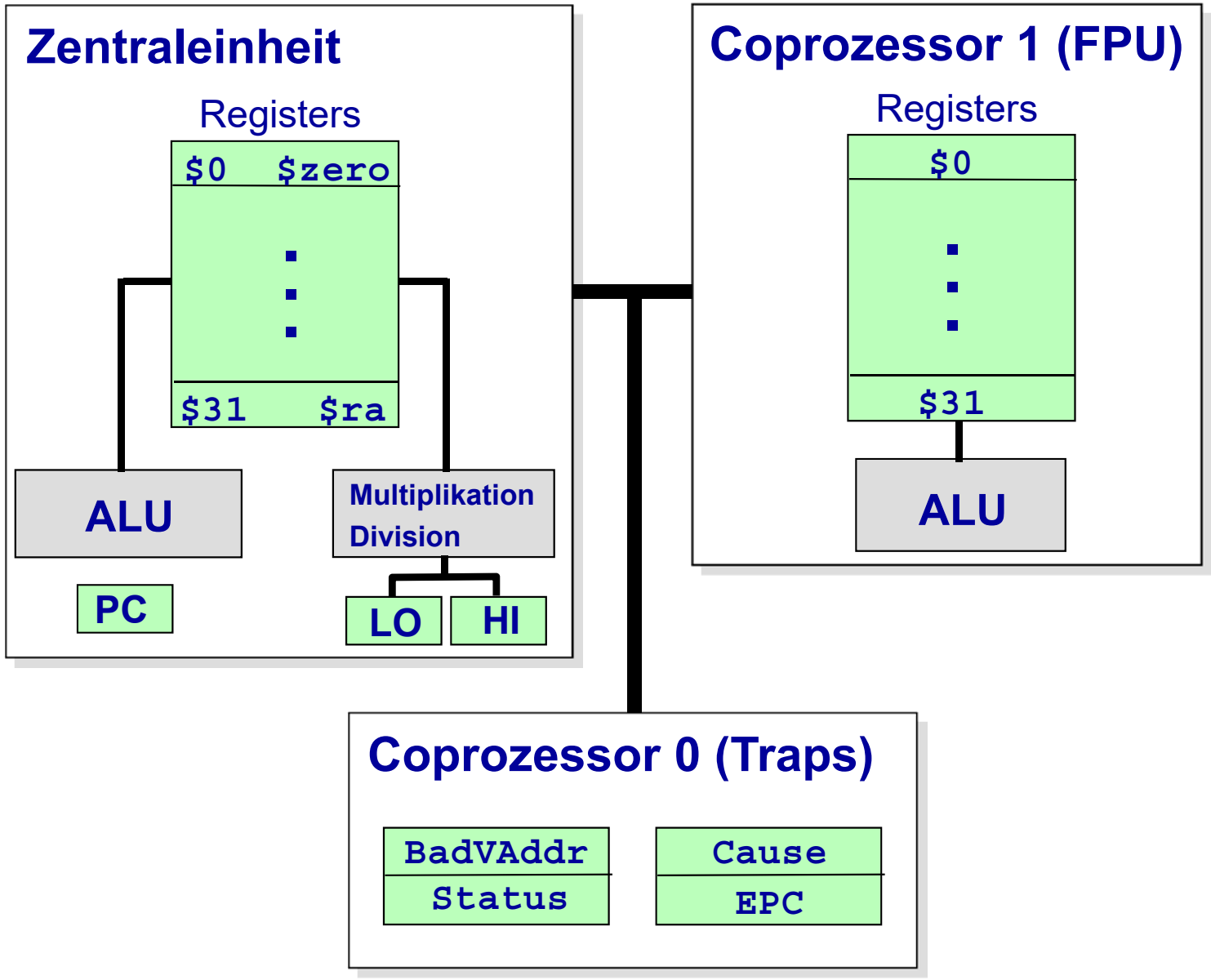
Reservierter Befehl (Code 10):

Es wurde versucht, einen Befehl auszuführen, dessen Bits 31...26 nicht als Opcode definiert sind

Arithmetischer Überlauf bei Ganzzahlberechnungen (Code 12):

Ergebnis einer `ADD`, `ADDI` oder `SUB` Instruktion lässt sich nicht als 32-Bit Zweierkomplement darstellen

Aufbau des MIPS-Prozessors

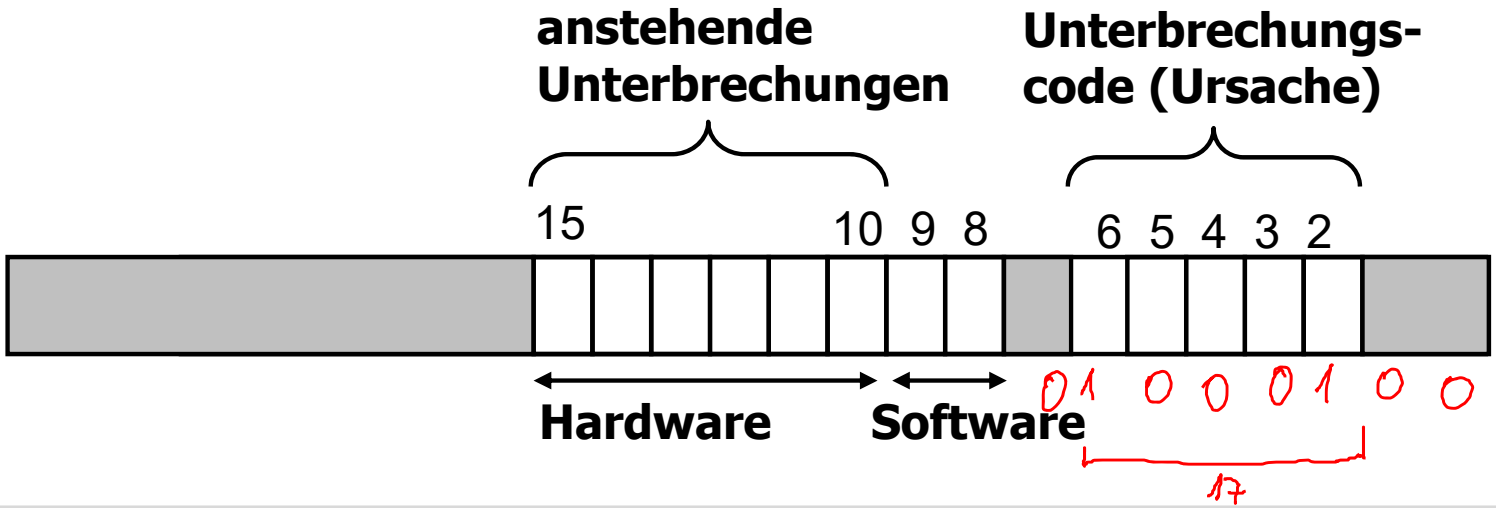


Register des Coprozessors 0

Coprozessor 0 enthält vier Register zur Behandlung von Ausnahmen und Unterbrechungen:

- **BadVAddr (Reg.-Nr. 8):**
  Falls die Unterbrechung durch einen Speicherzugriff verursacht wurde, ist hier die Speicheradresse enthalten

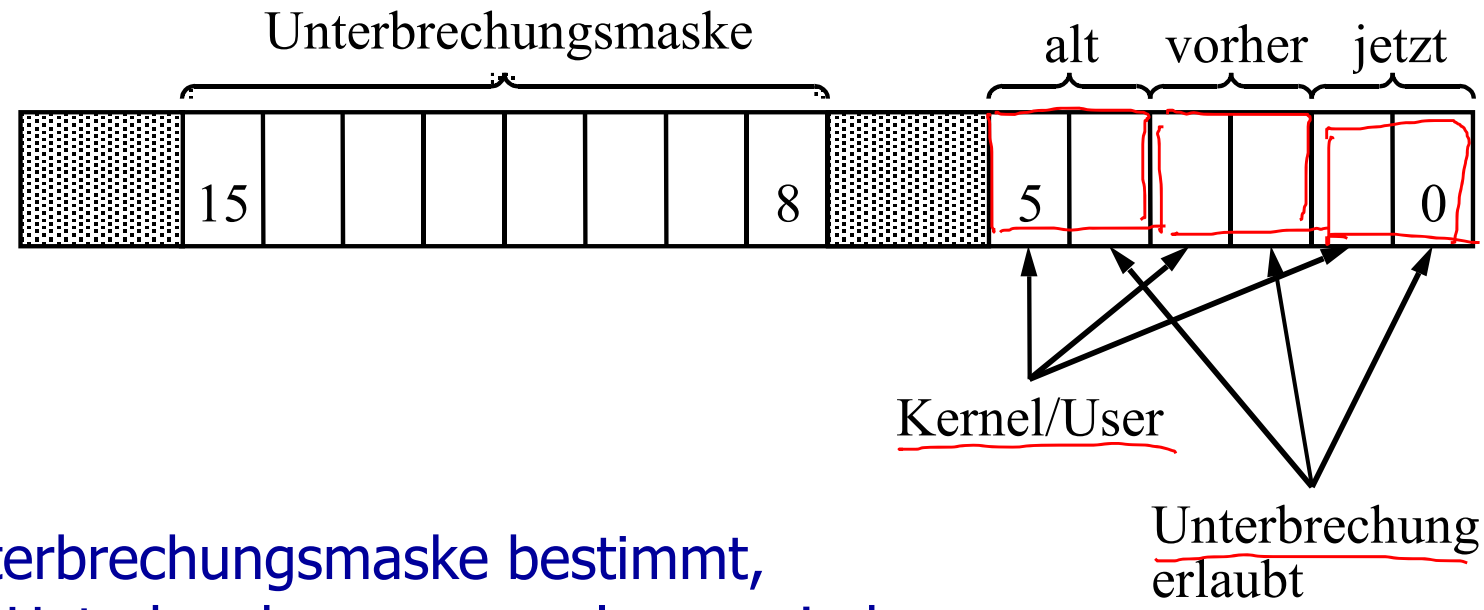
- **Cause (Reg.-Nr. 13):**



Register des Coprozessors 0

- **EPC (Reg.-Nr. 14):**
  Speicheradresse, in der sich der Befehl befindet, der zur Unterbrechung geführt hat

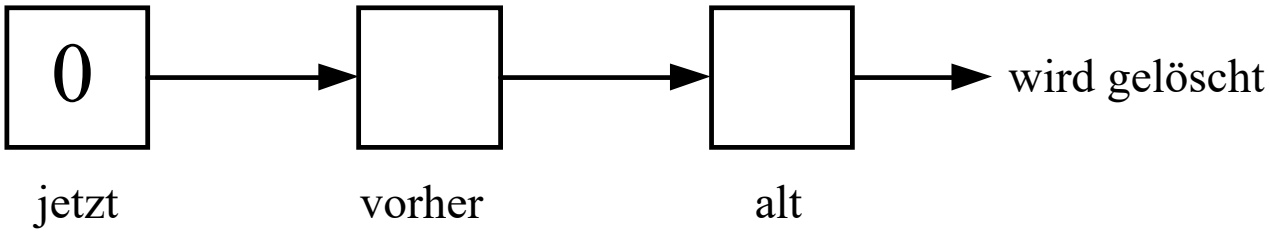
- **Status (Reg.-Nr. 12):**



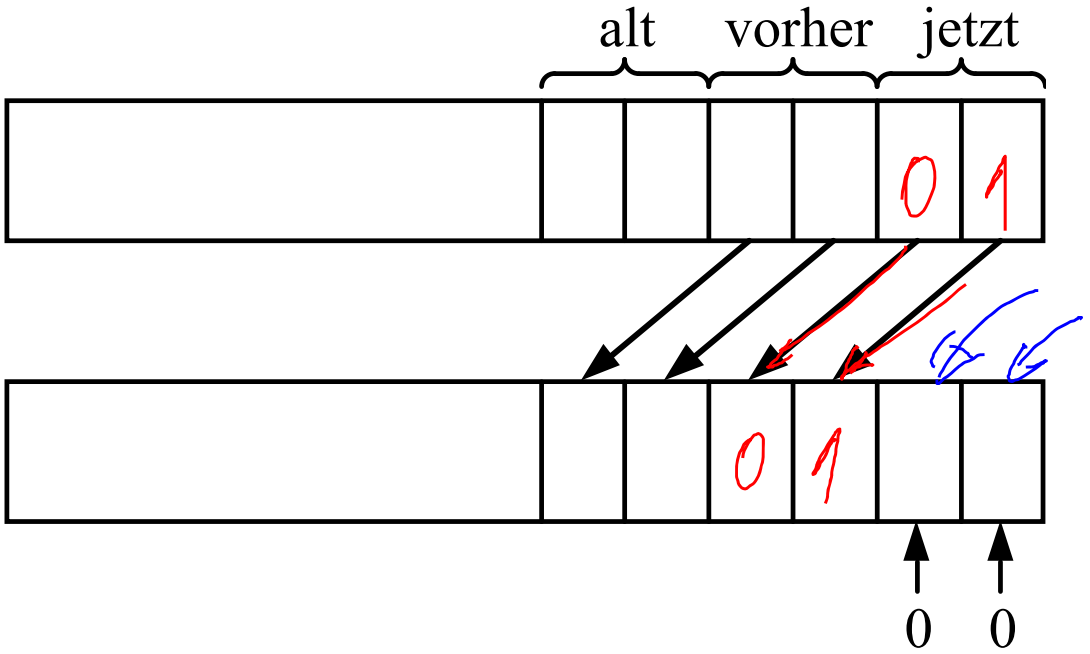
Die Unterbrechungsmaske bestimmt, welche Unterbrechungen zugelassen sind.

Register des Coprozessors 0

Speichern der Kernel/User- und Interrupt-Enable-Bits:

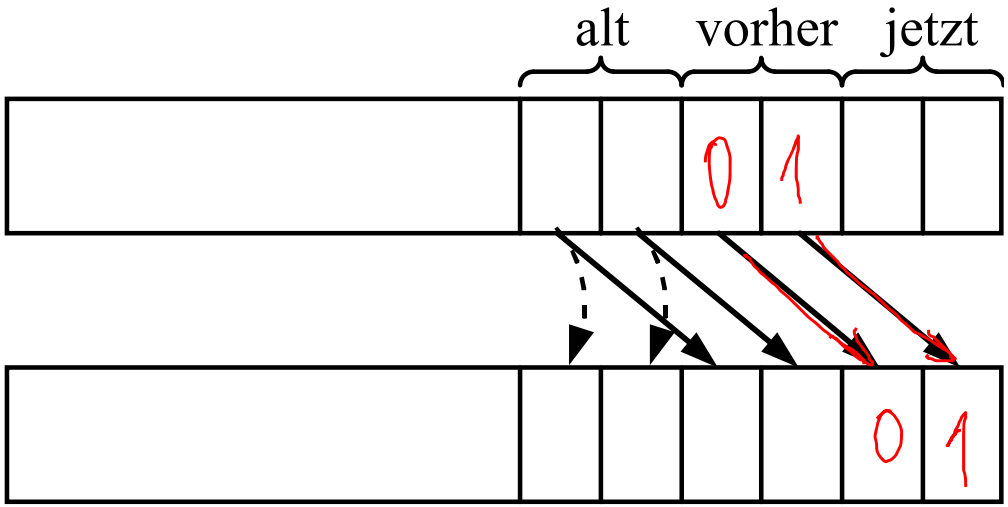


Beim Auftreten einer Unterbrechung:



Register des Coprozessors 0

Rücksprung aus einer Unterbrechung (`rfe` Instruktion):



Ausnahmebehandlung

- Man behandelt Unterbrechungen und Ausnahmen durch das Betriebssystem.
- Bei einer Unterbrechung oder Ausnahme erfolgt ein Sprung an eine festgelegte Stelle (bei MARS mit Standardkonfiguration: 0x8000 0180).
- Dort liegt eine Unterbrechungsbehandlungsroutine (Trap Handler), die die Unterbrechung bzw. Ausnahme behandelt.
- Die Ausführung einer Unterbrechungsbehandlungsroutine muss unter besonderen Betriebssystemrechten erfolgen, deshalb wird diese im Kernelsegment abgelegt.

Ausnahmebehandlung (Trap Handler)

Bei Ausnahme oder Unterbrechung erfolgt Sprung nach Adresse $8000\ 0180_{16}$:

```
.ktext 0x80000180
```

Benutzung des Stapels zur Sicherung von Daten nicht erlaubt, daher Sicherung von Registern im Kerneldatensegment:

```
sw $a0, a0_save
sw $a1, a1_save
sw $ra, ra_save
```

Register `$at` sichern:

```
.set noat
sw $at, at_save
.set at
```

Ausnahmebehandlung (Trap Handler)

Laden des Cause- und EPC-Registers:

```
mfc0 $k0, $13
mfc0 $k1, $14
```

Unterbrechungen ignorieren:

```
bgt $k0, 0x44, fertig
```

Spezifische Maßnahmen:

... (Benutzung von `$a0` und `$a1`)

```
jal print_excp
```

Abschluss der Ausnahmebehandlung

```
fertig:   lw $a0, a0_save
          lw $a1, a1_save
          lw $ra, ra_save
          .set noat
          lw $at, at_save
          .set at
```

Statusregister wiederherstellen:

```
          rfe
```

Fehlerhafte Instruktion soll nicht nochmals ausgeführt werden:

```
          addiu $k1, $k1, 4
```

Abschluss der Ausnahmebehandlung

Zurück zum Hauptprogramm: (nur falls nicht Ausnahme 6)

```
jr $k1
```

Kerneldatensegment:

```
        .kdata
a0_save:    .word 0
a1_save:    .word 0
ra_save:    .word 0
at_save:    .word 0
```

.set und sbrk

`.set (noat|at)`

Warnung des Assemblers über die Benutzung des `$at`-Registers ein bzw. ausschalten (in MARS nicht implementiert)

```
.set noat                    # Warnung ausschalten
 lw $at, at_save
 .set at                     # Warnung einschalten
```

sbrk (Syscall #9):

- Vergrößerung des Datensegments
- Dient zur dynamischen Speicherallokation