

```
FUN:  sw $ra,-4($sp)
      sw $s0,-8($sp) # $s0 für c
      sw $s1,-12($sp) # $s1 für j
      sw $s2,-16($sp) # $s2 für i
      sw $s3,-20($sp) # $s3 für m
      addi $sp,$sp,-20
      move $s0,$a0
      move $s1,$a1
      move $s2,$a3
      addi $s3,$a2,9
      move $a0,$s2
      move $a1,$s1
      jal V
      add $s3,$s3,$v0
      move $a0,$s1
      move $a1,$s2
      jal X
      add $s3,$s3,$v0
      sll $t0,$s3,2
      add $t0,$t0,$s0
      lw $v0,0($t0)
      addi $sp,$sp,20
      lw $ra,-4($sp)
      lw $s0,-8($sp) # $s0 für c
      lw $s1,-12($sp) # $s1 für j
      lw $s2,-16($sp) # $s2 für i
      lw $s3,-20($sp) # $s3 für m
      jr $ra
```

# Rechnerarchitektur (AIN 2)

## SoSe 2021

## Kapitel 2

### Befehle: Die Sprache des Rechners

Prof. Dr.-Ing. Michael Blaich  
mblaich@htwg-konstanz.de

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

**2.5 Kontrollstrukturen**

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.3 Prozeduren

2.5.4 Beispiel: Bubble-Sort

2.6 MIPS Assembler und MARS Simulator

...

# Kontrollstrukturen

## Alternative Pfade der Programmausführung

- Bedingte Programmausführung

```
if ... then ... [else ...]  
switch ... case ...
```

- Wiederholte Programmausführung

```
for ...  
while ...
```

## Prozeduren und Funktionen

```
int square(int x) {  
    int square_of_x;  
    square_of_x = x * x;  
    return square_of_x;  
}
```

```
int a, b;  
a=square(b);
```

# Programmzähler

- Der Programmzähler (Program Counter, PC) zeigt auf die auszuführende Instruktion
- Der Programmzähler wird als Speicheradresse der auszuführenden Instruktion im Register **PC** gespeichert

Program  
Counter  
(PC)



Speicher- adresse	Instruktion (hex)	(code)	Kommentar
[00400000]	8fa40000	lw \$4, 0(\$29)	; 183: lw \$a0 0(\$sp) # argc
[00400004]	27a50004	addiu \$5, \$29, 4	; 184: addiu \$a1 \$sp 4 # argv
[00400008]	24a60004	addiu \$6, \$5, 4	; 185: addiu \$a2 \$a1 4 # envp
[0040000c]	00041080	sll \$2, \$4, 2	; 186: sll \$v0 \$a0 2
[00400010]	00c23021	addu \$6, \$6, \$2	; 187: addu \$a2 \$a2 \$v0

## *Zur Erinnerung*

- Programme liegen im Binärformat im Hauptspeicher
- MIPS Instruktion entsprechen einem Wort (32 Bit)
- Programme werden Befehl für Befehl abgearbeitet

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

**2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen**

2.5.2.1 Bedingte und unbedingte Sprünge

2.5.2.2 Instruktionen für Vergleiche

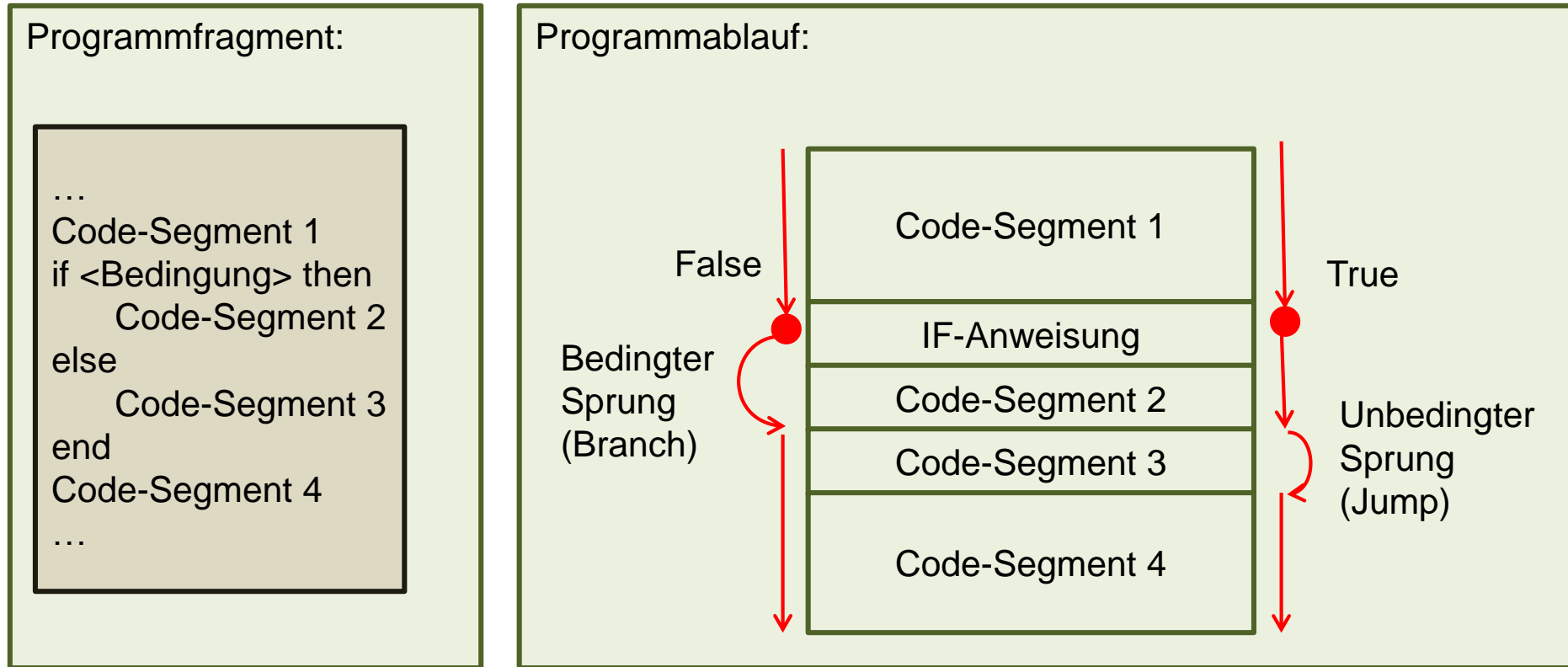
2.5.2.3 Jump-Table für SWITCH ... CASE

2.5.3 Prozeduren

...

# IF-Anweisung

## IF-Anweisungen und Schleifen implizieren nicht-sequentielle Ausführung von Anweisungen



# Sprungbefehle in MIPS

**Auf der Ebene der Maschinensprache gibt es keine „goto-lose“ (strukturierte) Programmierung!**

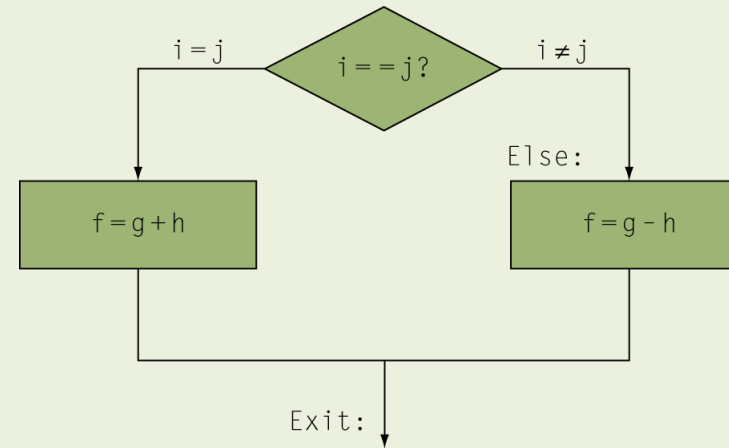
- Bedingte und unbedingte Sprunginstruktionen
  - `beq reg1, reg2, label` (branch if equal)
  - `bne reg1, reg2, label` (branch if not equal)
  - `j label` (jump to label/address)
- Label
  - „Label“ (Sprungmarke) werden in der Assemblersprache als Bezeichner für Programmzeilen genutzt
  - der Assembler übersetzt „Label“ in der Assemblersprache in „Sprungweiten“ (beq, bne) bzw. Adressen (j) im Binärformat

```
Label: add $t0, $s0, $s1
```

# IF-Anweisung: Ein Beispiel

C-Code:

```
if (i==j) {  
    f = g+h;  
}else {  
    f = g-h;  
}
```



Situation: Werte für Variablen  $f$ – $j$  in Registern  $\$s0$ – $\$s4$

Assembler-Code:

```
        bne $s3, $s4, Else    # gehen nach Else, wenn i != j  
        add $s0, $s1, $s2     # f=g+h (bei i != j übersprungen)  
        j    Exit            # gehe nach Exit  
Else:   sub $s0, $s1, $s2     # f=g-h (bei i == j übersprungen)  
Exit:
```



# WHILE-Schleife: Ein Beispiel

## C-Code:

```
while (safe[i] == k)
    i+=1;
```

## Situation:

- Basisadresse von Integer-Array `safe` in `$s6`
- Variablen `i` und `k` in `$s3` und `$s5`

## Schleife in Assembler:

- in jedem Schleifendurchlauf wird erst `safe[i]` geladen
- dann wird die Bedingung `safe[i]` ungleich `k` getestet und evtl. die Schleife beendet
- sonst wird `i` inkrementiert und zum Schleifenanfang gesprungen

## Assembler-Code:

```
Loop:    sll    $t1, $s3, 2        # Speicheroffset in Bytes ist i*4
         add    $t1, $t1, $s6      # Speicheradresse von safe(i)
         lw     $t0, 0($t1)        # $t0=safe(i)
         bne    $t0, $s5, Exit     # gehen nach Exit, wenn safe[i]!=k
         addi   $s3, $s3, 1        # i inkrementieren
         j      Loop              # gehe zu Loop

Exit:
```

# Adressierung in Jumps

## Bisher: Sprung zu Labels ... aber: Label in Maschinensprache?

- Jump in Maschinensprache – eine neues Format „J-Typ“

OP-Code (2)	Konstante (Sprungadresse)
6 Bits	26 Bits

- Sprungadresse ist die Speicheradresse der nächsten Instruktion und ist in „Worten“ angegeben (Achtung: in MARS in Bytes)

```
j    20000    # Setze PC auf 80000
           # 80000 ist die Speicheradresse in Bytes
```

- Adresse von Jump-Befehlen steht erst endgültig fest, wenn das Programm in den Hauptspeicher geladen wird
  - siehe Linker und Loader in Kapitel 2.8
  - Sprungadresse nur 26 Bit

# Adressierung in Branches

## Branches werden im I-Format kodiert

- 16 Bits sind nicht genug, um das ganze Programm zu Adressieren

OP-Code (5)	Register	Register	Konstante (Sprungweite)
6 Bits	5 Bits	5 Bits	16 Bits

- Sprungweite:

- in Branches wird keine Sprungadresse sondern eine Sprungweite spezifiziert

```
beq $s1, $s2, 100 # wenn $s1==$s2 setze PC=PC+400  
bne $s1, $s2, -50 # wenn $s1!=$s2 setze PC=PC-200
```

PC relative addressing –  
Adressierung relativ zum PC

- Die Sprungweite wird **relativ** vom PC bzw. der **Speicheradresse der nächsten Instruktion** gerechnet.
- Common Case Fast: Branches kommen hauptsächlich in IF-Anweisungen und Schleifen vor. Dort sind Sprünge meist kurz und mit 16 Bit adressierbar.
- **Sprungweite wird in Worten adressiert**


# Ein Beispiel

```
Loop:    sll    $t1, $s3, 2      # Speicheroffset in Bytes ist i*4
         add    $t1, $t1, $s6    # Speicheradresse von safe(i)
         lw     $t0, 0($t1)      # $t0=safe(i)
         bne    $t0, $s5, Exit   # gehen nach Exit, wenn safe[i]==$s5
         addi   $s3, $s3, 1      # i inkrementieren
         j      Loop            # gehe zu Loop

Exit:
```

Wir gehen davon aus, dass dieser Assembler-Code ab Adresse 80000 im Hauptspeicher liegt. Der Maschinen-Code sieht dann wie folgt aus:


80000:	0	0	19	9	2	0	sll \$t1, \$s3, 2
80004:	0	9	22	9	0	32	add \$t1, \$t1, \$s6
80008:	35	9	8	0			lw \$t0, 0(\$t1)
80012:	5	8	21	2			bne \$t0, \$s5, Exit
80016:	8	19	19	1			addi \$s3, \$s3, 1
80020:	2	20000					j Loop
80024:							Exit



# Beispiel

Sprungadressen/-weiten, wenn der Code bei Adresse 20000 steht?


20000:	0	0	19	9	2	0	sll \$t1, \$s3, 2
20004:	0	9	22	9	0	32	add \$t1, \$t1, \$s6
20008:	35	9	8	0			lw \$t0, 0(\$t1)
20012:	5	8	21	?			bne \$t0, \$s5, Exit
20016:	8	19	19	1			addi \$s3, \$s3, 1
20020:	2	?					j Loop
20024:							Exit



# Beispiel

Sprungadressen/-weiten, wenn der Code bei Adresse 20000 steht?

20000:	0	0	19	9	2	0	sll \$t1, \$s3, 2
20004:	0	9	22	9	0	32	add \$t1, \$t1, \$s6
20008:	35	9	8		0		lw \$t0, 0(\$t1)
20012:	5	8	21		2		bne \$t0, \$s5, Exit
20016:	8	19	19		1		addi \$s3, \$s3, 1
20020:	2				5000		j Loop
20024:							Exit



**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.2.1 Bedingte und unbedingte Sprünge

**2.5.2.2 Instruktionen für Vergleiche**

2.5.2.3 Jump-Table für SWITCH ... CASE

2.5.3 Prozeduren

...

# Vergleiche

## Vergleiche werden mit „Set Less Than“ gemacht

- Die Befehle lautet **slt** und **slti**
- Bei **slt** werden zwei Register verglichen und je nach Ergebnis wird im Ergebniss Register eine „1“ oder eine „0“ gesetzt
- Bei **slti** wird ein Register mit einer Konstanten verglichen

## Beispiele

```
slt $s0, $s1, $s2    # es wird "1" in Register $s0 geschrieben,  
                     # falls Inhalt von Register $s1 < Inhalt von Register $s2  
                     # andernfalls wird "0" in Register $s0 geschrieben
```

```
slti $s0, $s1, 42     # es wird "1" in Register $s0 geschrieben,  
                     # falls Inhalt von Register $s1 < 42  
                     # andernfalls wird "0" in Register $s0 geschrieben
```



# Vergleiche mit "Set Less Than"

## ■ Beispiel

Pseudo-Code:

```
if (0 <= $s0 < $s1) ...  
    else ...  
        goto OutOfBound
```

Assembler:

```
slti    $t0, $s0, 0           # $t0=1, wenn $s0<0  
bne $t0, $zero, OutOfBound    # wenn NOT($t0==0) goto OutOfBound  
                                   # d.h. $t0==1 bzw. $s0<0 goto  
                                   OutOfBound  
slt $t0, $s0, $s1             # $t0=1, wenn $s0<$s1  
beq $t0, $zero, OutOfBound    # wenn ($t0==0) goto OutOfBound  
...                           # $t0==0 ⇔ ($s0<$s1)==0 ⇔ $s0>=$s1  
OutOfBound:
```

# Vergleich: Unsigned und Signed

**Vergleiche können sowohl unsigned als auch signed durchgeführt werden**

- Die Befehle für den Vergleich von zwei Registern lauten dann **slt** und **sltu**
- Die Befehle für den Vergleich von Registern und Konstante lauten dann **slti** und **sltiu**

Beispiel

- Registerinhalte:

\$s0:	1111	1111	1111	1111	1111	1111	1111	1111	-1
\$s1:	0000	0000	0000	0000	0000	0000	0000	0001	1

- Vergleich  $\$s0 < \$s1$  (Signed):

`slt $t0, $s0, $s1    # $t0=1    -1 < 1`

- Vergleich  $\$s0 < \$s1$  (Unsigned):

`sltu $t0, $s0, $s1    # $t0=0     $2^{31}-2 < 1$`

# Umfrage: Sprung

Registerinhalte:

\$s1=0xFFFF FFFF

\$s2=0x0000 0001

Wird gesprungen oder nicht?

```
...  
    slt    $t0, $s1, $s2  
    bne    $t0, $zero, lab  
...  
lab:
```

```
...  
    sltu   $t0, $s1, $s2  
    beq    $t0, $zero, lab  
...  
lab:
```

# Umfrage: Sprung

Registerinhalte:

\$s1=0xFFFF FFFF

\$s2=0x0000 0001

Wird gesprungen oder nicht?

**\$t0=1**

```
...  
slt  $t0, $s1, $s2  
bne  $t0, $zero, lab
```

lab:



**\$t0=0**

```
...  
sltu $t0, $s1, $s2  
beq  $t0, $zero, lab
```

lab:



# Weitere Instruktionen

	Befehl	Opcode (o) Function (f)	Syntax	Operation
Vergleiche	slt	101010	f \$d, \$s, \$t	\$d = (\$s < \$t)
	sltu	101001	f \$d, \$s, \$t	\$d = (\$s < \$t)
	slti	001010	f \$d, \$s, i	\$t = (\$s < SE(i))
	sltiu	001001	f \$d, \$s, i	\$t = (\$s < SE(i))
Bedingte Sprünge (Branch)	beq	000100	o \$s, \$t, label	if (\$s == \$t) pc += i << 2
	bgtz	000111	o \$s, label	if (\$s > 0) pc += i << 2
	blez	000110	o \$s, label	if (\$s <= 0) pc += i << 2
	bne	000101	o \$s, \$t, label	if (\$s != \$t) pc += i << 2
Unbedingte Sprünge (Jump)	j	000010	o label	pc += i << 2
	jal	000011	o label	\$31 = pc; pc += i << 2
	jalr	001001	o labelR	\$31 = pc; pc = \$s
	jr	001000	o labelR	pc = \$s

# Quiz: IF ... THEN ... ELSE

Programmieren sie in Assembler

`C = minimum(A, B)`

- Geben sind    A in \$s0  
                  B in \$s1  
                  C in \$s2

- PSEUDO-CODE

- Assembler

# Quiz: IF ... THEN ... ELSE

Programmieren sie in Assembler

C = minimum(A, B)

- Geben sind    A in \$s0  
                  B in \$s1  
                  C in \$s2

## ■ PSEUDO-CODE

Variante 1:

```
1  if (a<b)
2    THEN c=a
3    ELSE c=b
```

Variante 2:

```
1  c=b
2  if (a<b)
3    THEN c=a
```

## ■ Assembler

```
1      slt $t0,$s0,$s1
2      bne $t0,$zero,ELSE
3      add $s2,$s0,$zero
4      j  END
5  ELSE:
6      add $s2,$s1,$zero
7  END:
```

```
1      add $s2,$s1,$zero
2      slt $t0,$s0,$s1
3      bne $t0,$zero,END
4      add $s2,$s0,$zero
5  END:
```

# Quiz: WHILE

```
while ($s1>$s2)  
...
```

```
loop: ...
```

```
continue:
```



# Quiz: WHILE

```
while ($s1>$s2)
```

```
...
```

```
loop:...
```

```
    beq $s1, $s2, continue    # if $s1==$s2 goto continue
```

```
    slt $t0, $s1, $s2        # $t0=($s1<$s2)
```

```
    bne $t0, $zero, continue  # if $t0!=0 goto continue
```

```
    j loop
```

```
continue:
```

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.2.1 Bedingte und unbedingte Sprünge

2.5.2.2 Instruktionen für Vergleiche

**2.5.2.3 Jump-Table für SWITCH ... CASE**

2.5.3 Prozeduren

...

# Jump Tables für Switch-Case Anweisungen

- Die Jump-Table befindet sich als Array im Hauptspeicher und enthalten die Speicheradressen zu den „Labels“.
- Der Befehl **jr reg** führt als nächstes die Instruktion aus, die an der in `reg` gespeicherten Speicherstelle liegt.

Pseudo-Code:

```
switch (k)
  case 0: f=1;
  case 1: f=g+h;
  case 2: f=g-h;
end
```

Register:

\$s0-\$s2:	f, g, h
\$s3:	k
\$s4:	Basisadresse der Jump Table

Speicher-  
adresse

Assembler-Code:

# Jump Tables für Switch-Case Anweisungen

- Die Jump-Table befindet sich als Array im Hauptspeicher und enthalten die Speicheradressen zu den „Labels“.

Label	Adresse
Label1	0x400100
Label2	0x400108
Label3	0x400116

- Der Befehl **jr reg** führt als nächstes die Instruktion aus, die an der in `reg` gespeicherten Speicherstelle liegt.

Pseudo-Code:

```
switch (k)
  case 0: f=1;
  case 1: f=g+h;
  case 2: f=g-h;
end
```

Register:

\$s0-\$s2: f, g, h  
\$s3: k (0, 1 oder 2)  
\$s4: Basisadresse  
der Jump Table

Speicher-  
adresse

Assembler-Code:

```
                                sll    $t0, $s3, 2
                                add    $t0, $t0, $s4
                                lw     $t1, 0($t0)
                                jr     $t1
0x400100 Label1: addi    $s0, $zero, 1
                                j      Exit
0x400108 Label2: add    $s0, $s1, $s2
                                j      Exit
0x400116 Label3: sub    $s0, $s1, $s2
                                Exit:
```

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

...

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

**2.5.3 Prozeduren**

**2.5.3.1 Prinzip: Rücksprungadresse, Parameterübergabe und Stack**

2.5.3.2 Realisierung in MIPS

2.5.3.3 Verwendung des Stack

2.5.3.4 Beispiel: Fakultät

2.5.3.5 Speicherkonventionen in MIPS

...

# Prozeduren

## HW-Unterstützung für Prozeduren

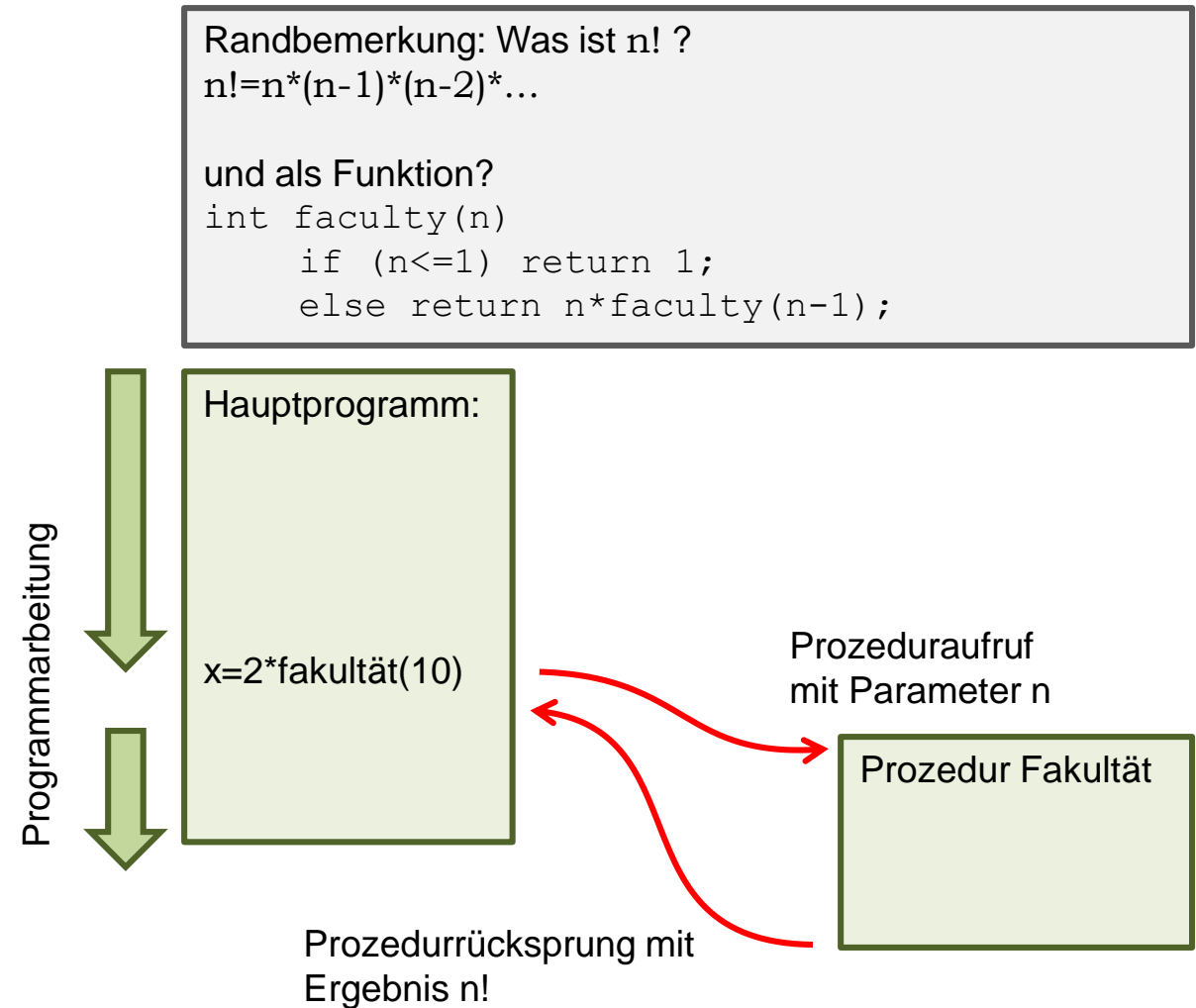
- Prozeduren stellen das wichtigste Strukturierungskonzept in modernen Programmiersprachen dar.
- Alle modernen Instruktionssatzarchitekturen bieten Mittel zur effizienten Bearbeitung von Prozeduren an.

## Schritte bei der Ausführung einer Prozedur

- Argumente so platzieren, dass die Prozedur darauf zugreifen kann
- Kontrolle an die Prozedur übergeben
- Prozedur ausführen
- Ergebniswert so platzieren, dass aufrufendes Programm darauf zugreifen kann
- Kontrolle an die Aufrufstelle zurück geben

## Prozeduren liegen an einer anderen Stelle im Speicher als das Hauptprogramm

- Springen können wir, aber was ist mit Parameterübergabe etc.



# Erstes Problem: Programmzähler und Rücksprungadresse

Rücksprungadresse wird in Register \$ra (return address) gespeichert

- Rekursion?

Startadresse des Hauptprogramms

Register \$pc	Register \$ra
0x0040008	-
...	
0x0040010	-
0x0040020	0x0040014
...	
0x004002c	0x0040014
0x0040014	-

Adresse	Binäre Instruktion	
0x0040000	: 0011 ... 1001	
0x0040004	: 0001 ... 1000	
0x0040008	: 1001 ... 1111	
0x004000c	: 1011 ... 0001	
0x0040010	: 0011 ... 1000	Aufruf der Prozedur
0x0040014	: 1001 ... 1111	Zurück
0x0040018	: 0001 ... 0001	
0x004001c	: 1011 ... 0011	
0x0040020	: 1011 ... 1100	Prozedur Fakultät
0x0040024	: 0101 ... 1001	
0x0040028	: 1000 ... 0011	
0x004002c	: 1000 ... 1011	Rücksprung aus der Prozedur
0x0040030	: 0001 ... 1100	
0x0040034	: 1001 ... 1111	
0x0040038	: 1001 ... 1111	

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

...

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

**2.5.3 Prozeduren**

2.5.3.1 Prinzip: Rücksprungadresse, Parameterübergabe und Stack

**2.5.3.2 Realisierung in MIPS**

2.5.3.3 Verwendung des Stack

2.5.3.4 Beispiel: Fakultät

2.5.3.5 Speicherkonventionen in MIPS

...



# Beispiel mit Parameterübergabe

Dieses Assemblerbeispiel dient dazu, die Sprungbefehle jal zum Prozeduraufruf und jr zum Rücksprung zu illustrieren sowie die benötigten Register \$pc (program counter), \$ra (return address), \$a0 (argument) und \$v0(result) einzuführen.

## Hauptprogramm:

```
0x004000c      addi  $a0, $zero, 10      # x=2*fakultät(10)
0x0040010      jal   Fakultaet          # Funktionsargument 10 in $a0
0x0040014      sll   $v0, $v0, 1        # rufe Prozedur auf
...                                     # berechne Rückgabe*2
```

## Prozedur Fakultät:

```
0x0040024      ...                      # Berechnen von n! und
...            ...                      # in $a0 schreiben
0x004002c      add   $v0, $a0, $zero    # Rückgabewert n! in $v0 schreiben
0x0040030      jr    $ra                # zurückspringen
...
```

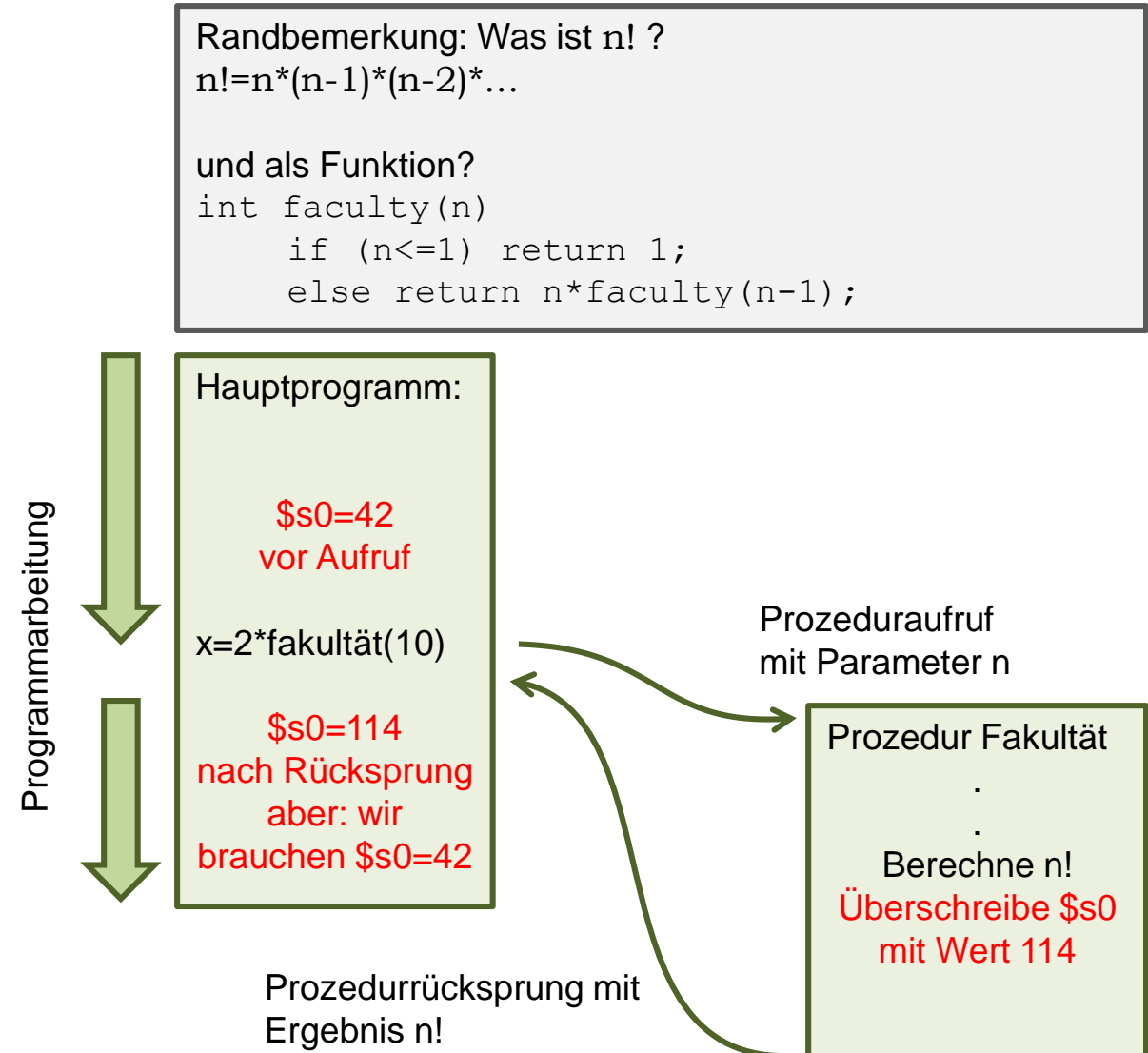
Register	\$pc						\$ra						\$a0						\$v0					
Inhalt	0c	10	24	2c	30	14	x	x	14	14	14	14	10	10	10	10!	10!	10!	x	x	x	10!	10!	2*10!

Hinweis: in \$pc und \$ra jeweils das letzte Byte der Speicheradresse der Instruktion

# Problem: Überschreiben von Registern in der Prozedur

## In Prozeduren verwenden wir zur Berechnung lokale Variablen

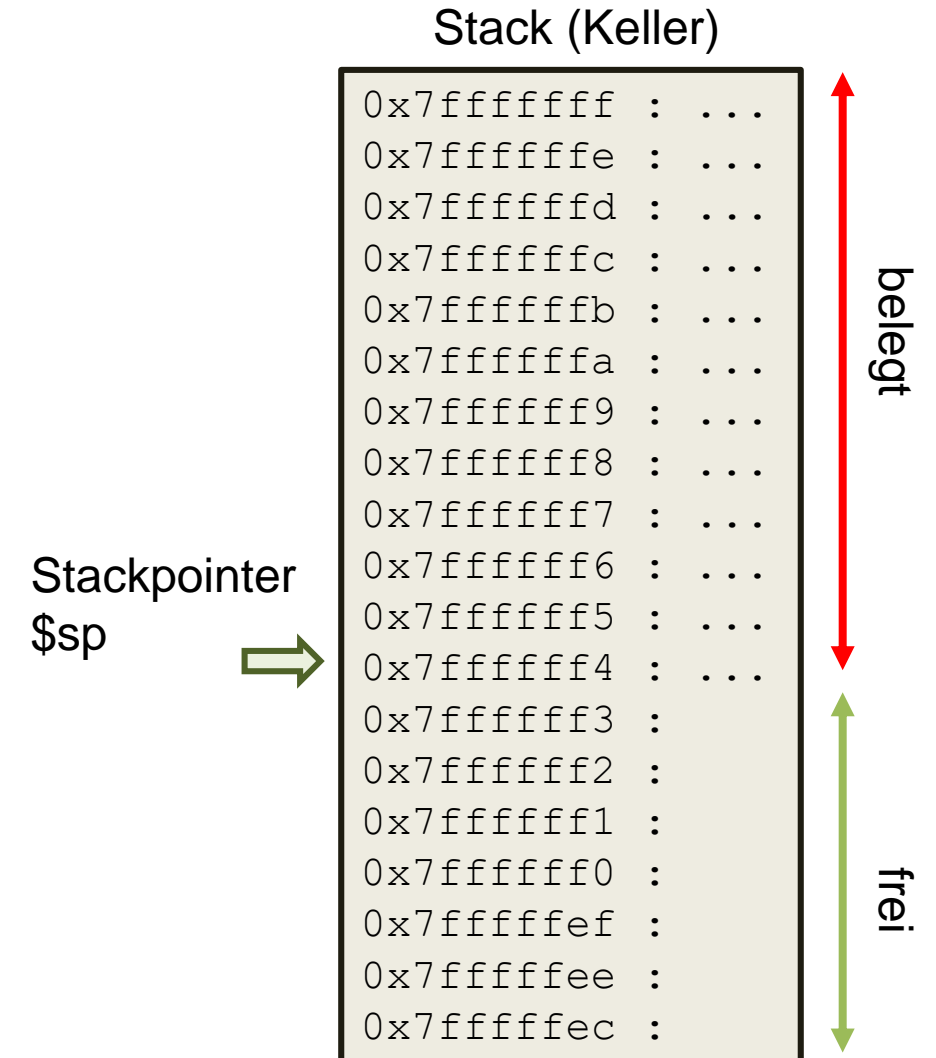
- Wir haben aber keine „lokalen“ Register
- Wenn die Prozedur einen Wert in ein Register schreibt, geht der bisherige Wert für das Hauptprogramm verloren
- Die Registerwerte müssen beim Prozeduraufruf gesichert und beim Rücksprung wieder hergestellt werden



# Stack

**Der Stack ist ein spezieller Speicherbereich, der Prozeduren zur Verfügung steht**

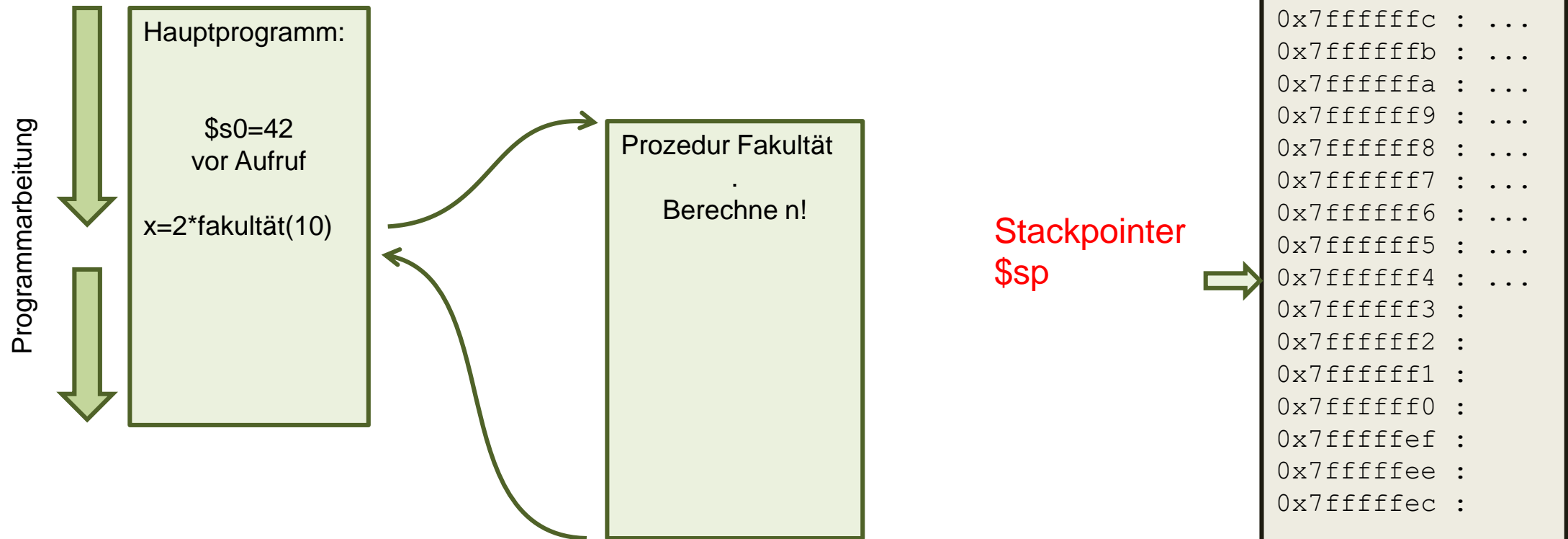
- Eine Prozedur darf den Stack vom Stackpointer an abwärts nutzen
- Unter anderem wird der Stack genutzt, um darauf Register zu sichern und sie nach Ablauf der Prozedur wieder herzustellen.
- Der Stackpointer steht im Register \$sp und zeigt auf den letzten im Stack abgelegten Wert, d.h. die niedrigste im Stackspeicher (sinnvoll) belegte Adresse.



# Problem: Überschreiben von Registern in der Prozedur

## Situation vor Aufruf der Prozedur

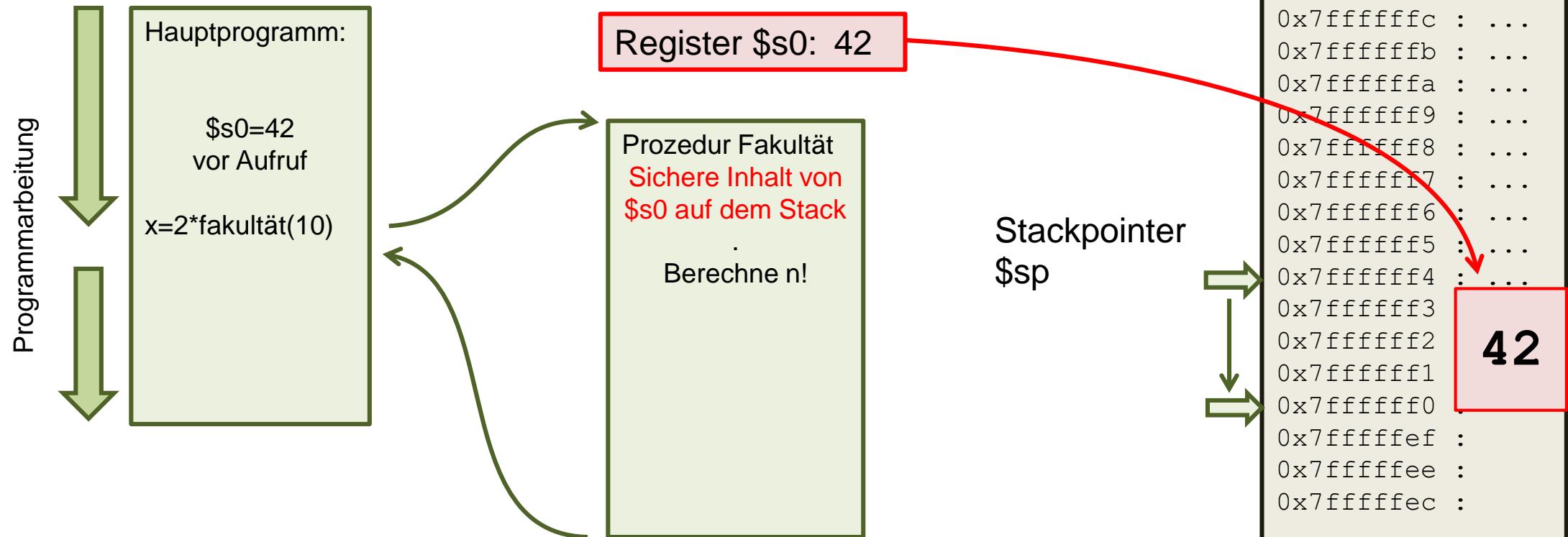
- Der Stackpointer steht im Register `$sp` und zeigt auf den letzten im Stack abgelegten Wert, d.h. die niedrigste im Stackspeicher (sinnvoll) belegte Adresse.



# Problem: Überschreiben von Registern in der Prozedur

## Am Anfang der Prozedur: Sichern der Register

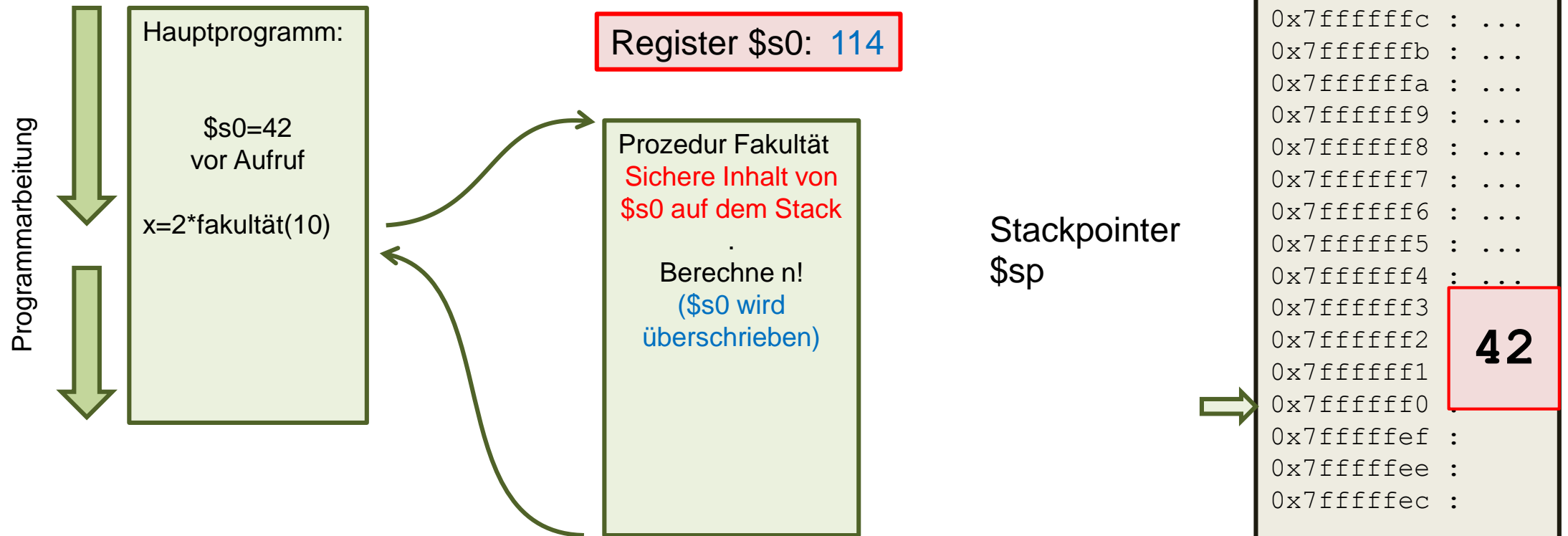
- Die in der Prozedur benutzte \$s-Register auf dem Stack sichern.
- Der Stackpointer wird pro gesichertem Register um 4 Bytes verringert, so dass er wieder auf die niedrigste belegte Adresse im Stackspeicher zeigt.



# Problem: Überschreiben von Registern in der Prozedur

## Während der Ausführung der Prozedur

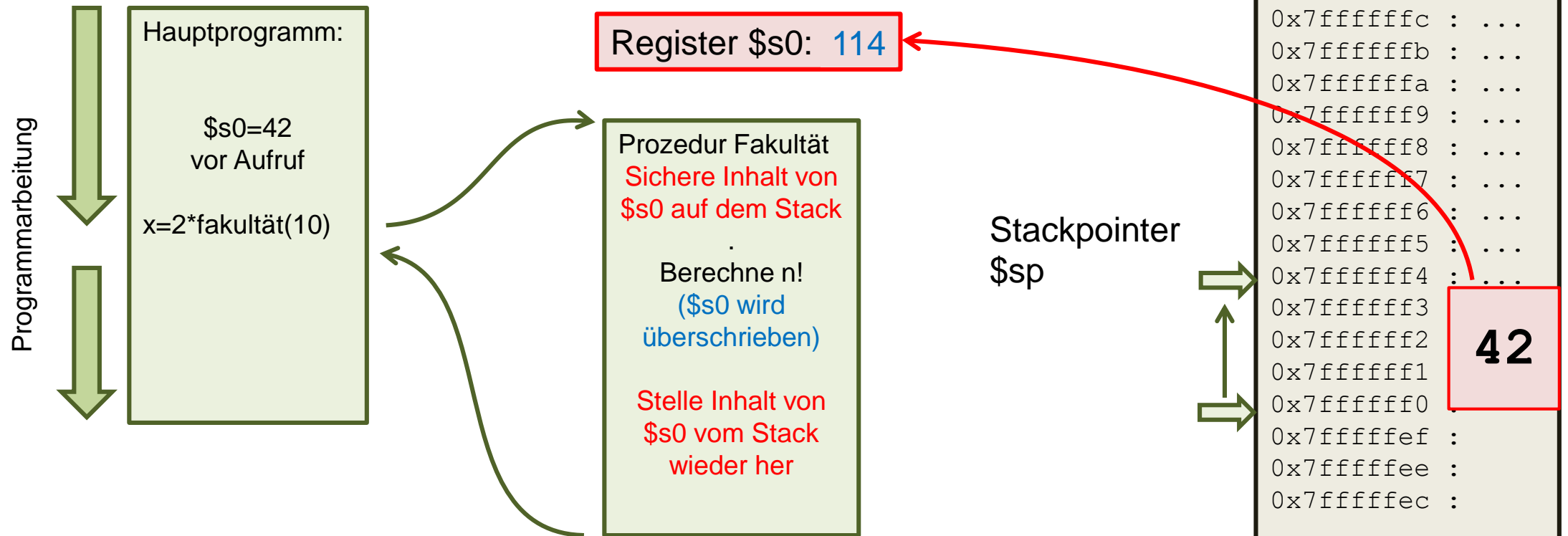
- Die \$s-Register werden benutzt und überschrieben



# Problem: Überschreiben von Registern in der Prozedur

## Am Ende der Prozedur

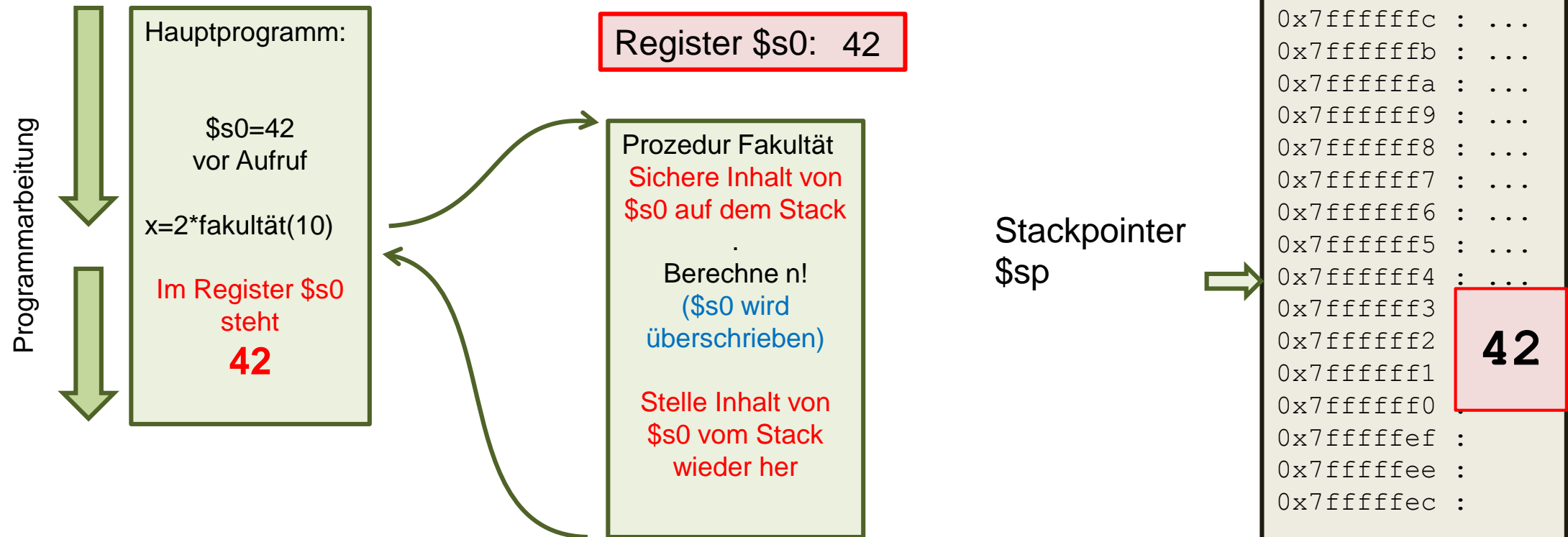
- Der ursprüngliche Inhalt der \$s-Register wird vom Stack wieder hergestellt
- Der Platz auf dem Stack wird wieder freigegeben, indem auch der Stackpointer auf den Stand zum Prozeduraufruf zurückgesetzt wird



# Problem: Überschreiben von Registern in der Prozedur

## Nach Rückkehr aus der Prozedur

- Inhalte der \$s-Register und des \$sp auf dem Stand wie vor dem Aufruf
- Vereinbarungsgemäß liegt die Verantwortung dafür bei der Prozedur
- Das Hauptprogramm kann mit seiner Berechnung fortfahren





# Sichern der Register in Assembler

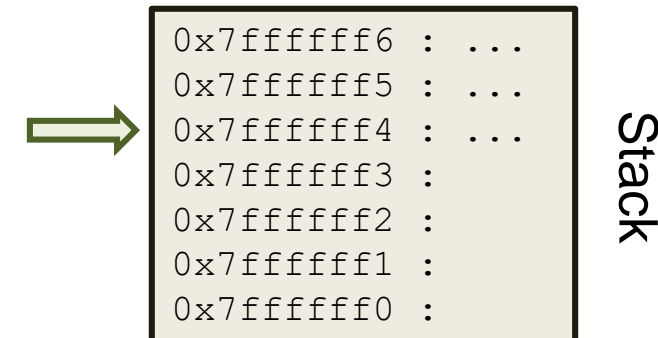
## Prozedur Fakultät:

```
Fakultaet: addi    $sp, $sp, -4    # Stackpointer um ein Wort verringern
            sw     $s0, 0($sp)    # $s0 auf den Stack schreiben

            ...                    # Berechne n!

            lw     $s0, 0($sp)    # $s0 vom Stack wiederherstellen
            addi    $sp, $sp, 4    # Stackpointer um ein Wort erhöhen
```

\$sp	\$s0
0x7fffffff4	42



# Sichern der Register in Assembler

Prozedur Fakultät:

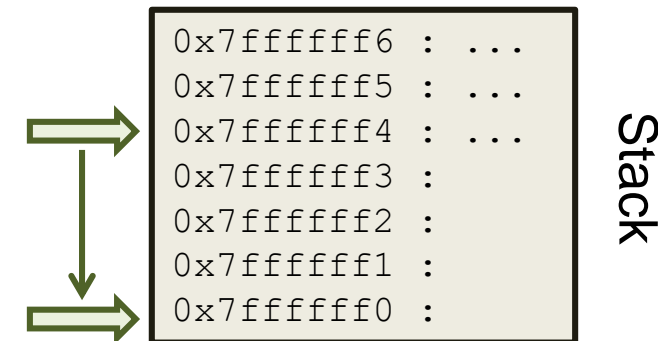
```
Fakultaet: addi $sp, $sp, -4    # Stackpointer um ein Wort verringern
            sw    $s0, 0($sp)   # $s0 auf den Stack schreiben

            ...                  # Berechne n!

            lw    $s0, 0($sp)   # $s0 vom Stack wiederherstellen
            addi $sp, $sp, 4     # Stackpointer um ein Wort erhöhen
```

- Verringern des Stackpointers, um auf dem Stack Platz zu schaffen. Der Stackpointer muss um 4 Bytes für jedes zu sicherndes Register verringert werden.

\$sp	\$s0
0x7fffffff4	42
0x7fffffff0	42



# Sichern der Register in Assembler

## Prozedur Fakultät:

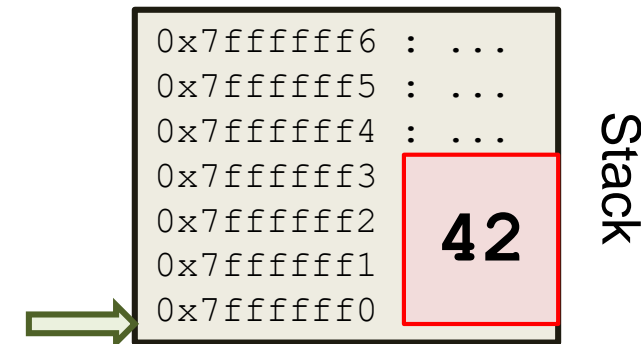
```
Fakultaet: addi  $sp, $sp, -4    # Stackpointer um ein Wort verringern
            sw    $s0, 0($sp)    # $s0 auf den Stack schreiben

            ...                  # Berechne n!

            lw    $s0, 0($sp)    # $s0 vom Stack wiederherstellen
            addi  $sp, $sp, 4     # Stackpointer um ein Wort erhöhen
```

- Verringern des Stackpointers, um auf dem Stack Platz zu schaffen. Der Stackpointer muss um 4 Bytes für jedes zu sicherndes Register verringert werden.
- Register \$s0 auf Stack speichern.

\$sp	\$s0
0x7fffffff4	42
0x7fffffff0	42



# Sichern der Register in Assembler

## Prozedur Fakultät:

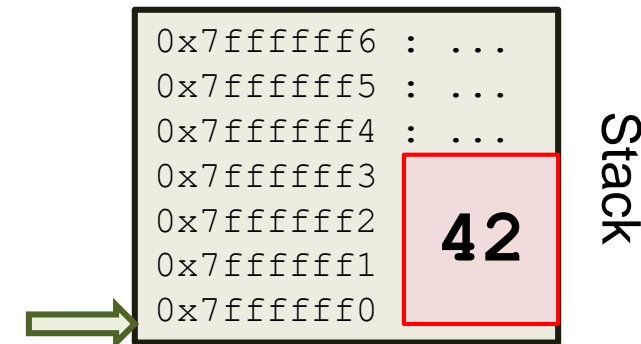
```
Fakultaet: addi  $sp, $sp, -4    # Stackpointer um ein Wort verringern
            sw    $s0, 0($sp)    # $s0 auf den Stack schreiben

            ...                  # Berechne n!

            lw    $s0, 0($sp)    # $s0 vom Stack wiederherstellen
            addi  $sp, $sp, 4     # Stackpointer um ein Wort erhöhen
```

- Verringern des Stackpointers, um auf dem Stack Platz zu schaffen. Der Stackpointer muss um 4 Bytes für jedes zu sicherndes Register verringert werden.
- Register \$s0 auf Stack speichern.
- Im Verlauf der Prozedur kann das Register \$s0 genutzt und der Inhalt des Registers verändert werden.

\$sp	\$s0
0x7fffffff4	42
0x7fffffff0	42
0x7fffffff0	114



# Sichern der Register in Assembler

## Prozedur Fakultät:

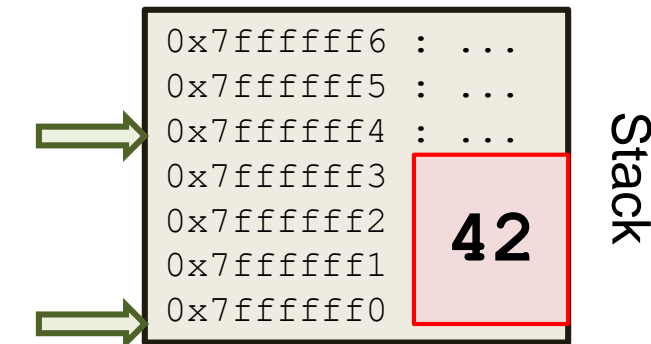
```
Fakultaet: addi  $sp, $sp, -4    # Stackpointer um ein Wort verringern
            sw    $s0, 0($sp)    # $s0 auf den Stack schreiben

            ...                  # Berechne n!

            lw    $s0, 0($sp)    # $s0 vom Stack wiederherstellen
            addi  $sp, $sp, 4     # Stackpointer um ein Wort erhöhen
```

- Verringern des Stackpointers, um auf dem Stack Platz zu schaffen. Der Stackpointer muss um 4 Bytes für jedes zu sicherndes Register verringert werden.
- Register \$s0 auf Stack speichern.
- Im Verlauf der Prozedur kann das Register \$s0 genutzt und der Inhalt des Registers verändert werden.
- Der ursprüngliche Inhalt des Registers \$s0 wird vom Stack geladen und wiederhergestellt.

\$sp	\$s0
0x7fffffff4	42
0x7fffffff0	42
0x7fffffff0	114
0x7fffffff0	42



# Sichern der Register in Assembler

## Prozedur Fakultät:

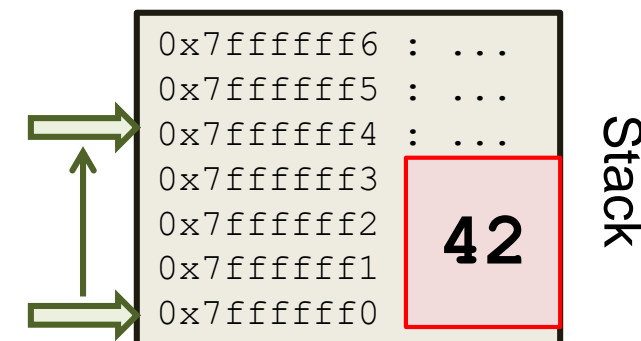
```
Fakultaet: addi $sp, $sp, -4    # Stackpointer um ein Wort verringern
           sw   $s0, 0($sp)    # $s0 auf den Stack schreiben

           ...                  # Berechne n!

           lw   $s0, 0($sp)    # $s0 vom Stack wiederherstellen
           addi $sp, $sp, 4     # Stackpointer um ein Wort erhöhen
```

- Verringern des Stackpointers, um auf dem Stack Platz zu schaffen. Der Stackpointer muss um 4 Bytes für jedes zu sicherndes Register verringert werden.
- Register \$s0 auf Stack speichern.
- Im Verlauf der Prozedur kann das Register \$s0 genutzt und der Inhalt des Registers verändert werden.
- Der ursprüngliche Inhalt des Registers \$s0 wird vom Stack geladen und wiederhergestellt.
- Der auf dem Stack belegte Speicherplatz wird wieder freigegeben, indem der Stackpointer um die entsprechende Anzahl Bytes – 4 pro gesichertem Register – wieder erhöht wird.

\$sp	\$s0
0x7fffffff4	42
0x7fffffff0	42
0x7fffffff0	114
0x7fffffff0	42
0x7fffffff4	42



# Zusammenfassung GNU-MIPS-Unterstützung von Prozeduren

## Übersicht der Register und Sprungbefehle, die bei der Durchführung von Prozeduren genutzt werden

### ■ Zuordnungen von Registern

`$a0 - $a3:`      Argumente  
`$v0, $v1:`      Rückgabewerte  
`$ra:`              Rücksprungadresse

### ■ Befehl für den Aufruf

`jal Prozedur (jump and link)`  
– Spezialinstruktion für Aufruf von Prozeduren  
– Springt zu Prozedur  
– speichert Adresse der folgenden Instruktion in `$ra`: `$ra = PC + 4`

### ■ Befehl für den Rücksprung

`jr $ra`  
– Gewöhnlicher Sprungbefehl  
– liest Sprungziel aus `$ra`

## Übersicht der Verwendung und Verwaltung des Stacks

- Neben Registern können auch weitere Argumente auf den Stack gelegt werden, falls die 4 Register \$a0-\$a3 nicht ausreichen.
- Argumentspeicher und Zustandssicherung
  - Im Speicher wird ein Stack verwaltet, der dynamisch die Prozedurverwaltung unterstützt.
  - Er enthält
    - Argumentwerte, wenn mehr als 4 auftreten
    - Register, die vor Prozeduraufruf gesichert wurden.
- Stackverwaltung
  - Ein Register weist auf den „Top of the Stack“, d.h. die niedrigste belegte Adresse im Stack (Stackpointer \$sp)
  - Der Stack ist am „oberen Ende“ des Speichers angeordnet und wächst in Richtung kleinerer Speicheradressen.



# Übersicht der Register

- Die aufrufende Prozedur sichert nur die Register \$s0-\$s7, den Stackpointer und die Rücksprungadresse.
- Was ist mit den anderen Registern?
- Was ist bei rekursiven Aufrufen?

Name	Nummer	Verwendung	Wird über Aufrufgrenzen bewahrt?
\$zero	0	Konstante 0	n.a.
\$at	1		nein
\$v0-\$v1	2-3	Prozedur-Rückgabe	nein
\$a0-\$a3	4-7	Prozedur-Parameter	nein
\$t0-\$t7	8-15	Temporäre	nein
\$s0-\$s7	16-23	Temporär gesicherte	ja
\$t8-\$t9	24-25	Temporäre	nein
\$k0-\$k1	26-27		nein
\$gp	28		ja
\$sp	29	Stack-Pointer	ja
\$fp	30		ja
\$ra	31	Return-Adresse	ja

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

...

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

**2.5.3 Prozeduren**

2.5.3.1 Prinzip: Rücksprungadresse, Parameterübergabe und Stack

2.5.3.2 Realisierung in MIPS

**2.5.3.3 Verwendung des Stack**

2.5.3.4 Beispiel: Fakultät

2.5.3.5 Speicherkonventionen in MIPS

...

# Rekursiver Aufruf von Prozeduren

## Prozeduren liegen an einer anderen Stelle im Speicher

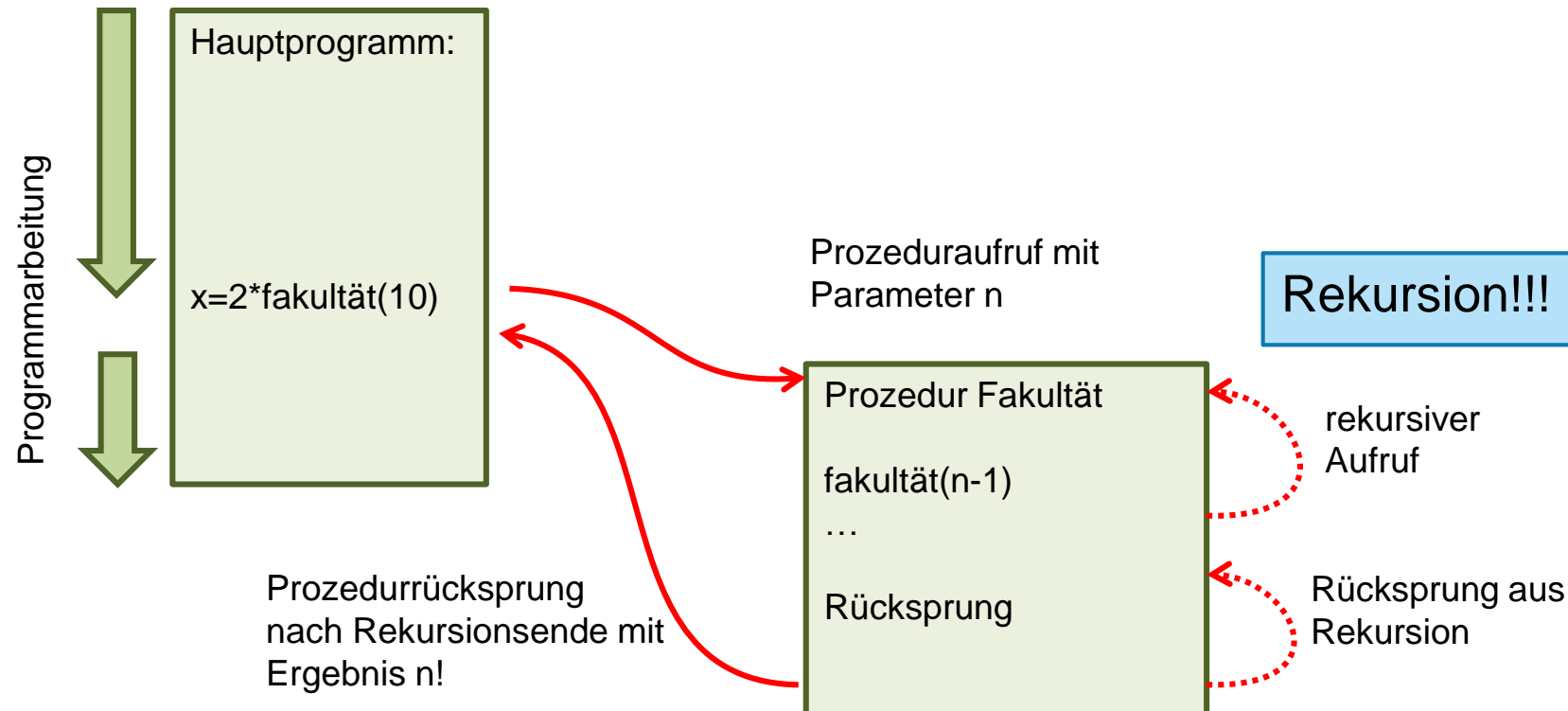
- Springen können wir aber was ist mit Parameterübergabe etc.

Randbemerkung: Was ist  $n!$  ?

$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots$

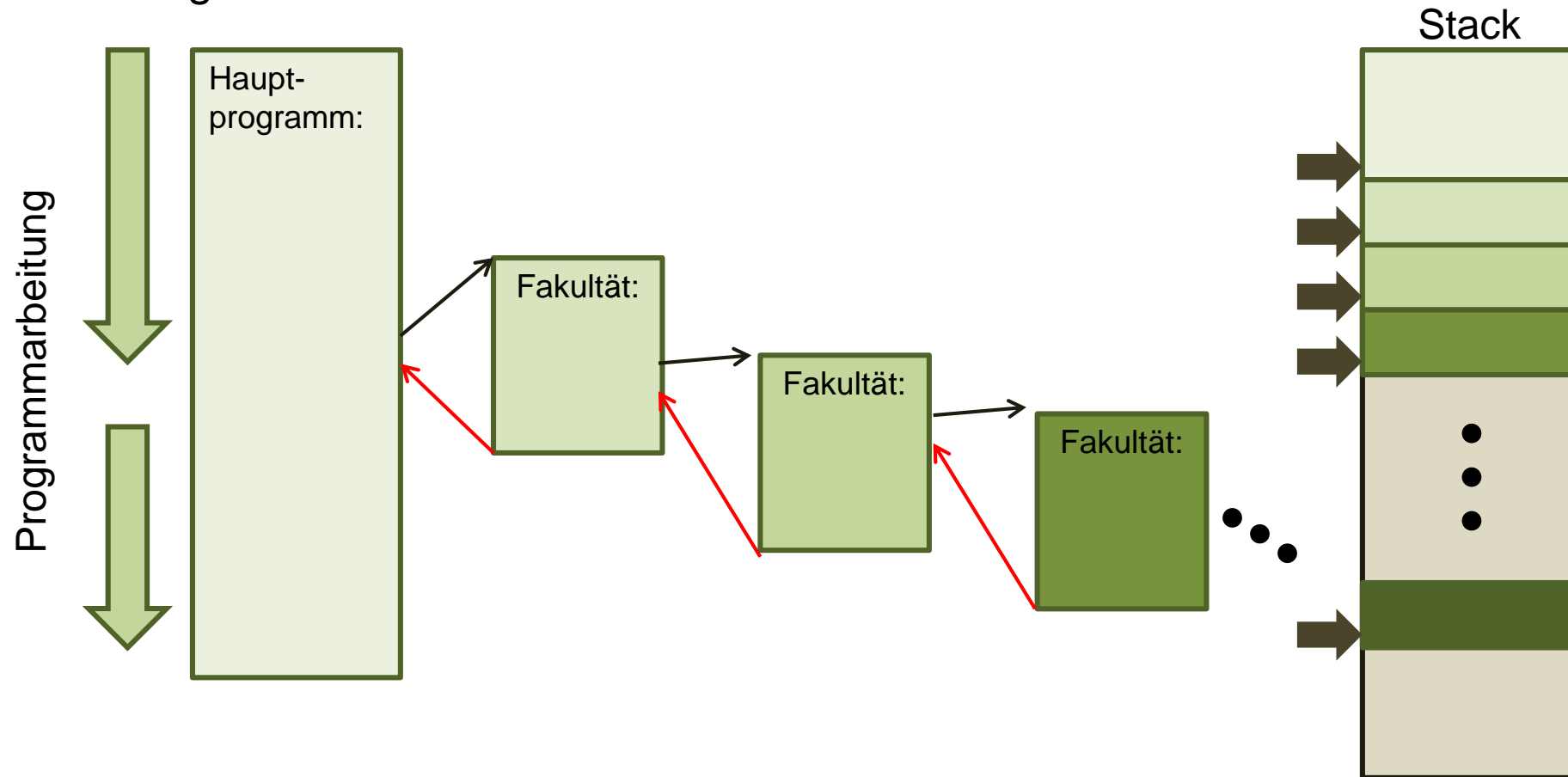
und als Funktion?

```
int fakultät(n)
    if (n<=1) return 1;
    else return n*fakultät(n-1);
```



# Verwendung des Stacks bei Rekursionen

- Der Stack wächst von oben nach unten. Eine Prozedur greift nicht auf die „weiter oben liegenden“ Teile des Stack zu und hinterlässt den „oberen Teil des Stack“ so, wie er beim Aufruf vorgefunden wurde.



# Was ist zu sichern und wer sichert?

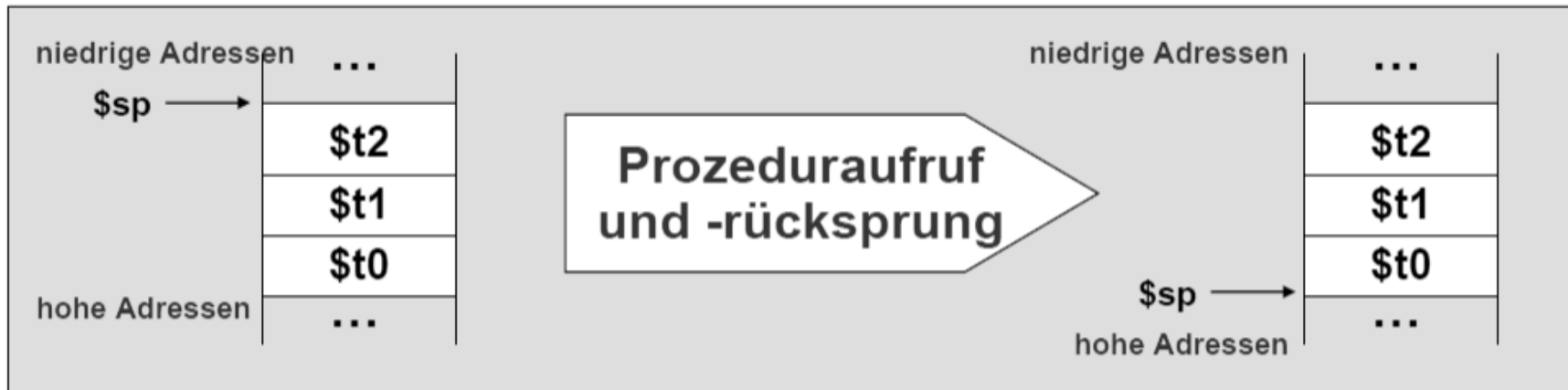
Es muss eindeutig spezifiziert sein, wer für die Sicherung von Registern zuständig ist: die aufrufende oder die aufgerufene Funktion.

- Probleme
  - Alle Prozeduren verwenden `$a0–$a3` für Argumente
  - Alle Prozeduren verwenden `$ra` als Rücksprungregister
  - Alle Prozeduren verwenden die Arbeitsregister `$s0–$s7` und `$t0–$t9`
- Wer sichert was (eine Möglichkeit)?
  - Aufrufende Prozedur sichert Argumentregister auf Stack
  - Aufrufende Prozedur sichert *temporäre* Arbeitsregister auf Stack:  
*temporäre Register: \$t0–\$t9*
  - Aufgerufene Prozedur sichert Rücksprungregister `$ra` und verwendete gespeicherte Register auf Stack  
*gespeicherte Register: \$s0–\$s7*
  - Vor **Rücksprung** aus einer Prozedur werden Rücksprungregister `$ra` und gespeicherte Arbeitsregister `$s*` vom Stack geladen.
  - Nach **Rücksprung** aus aufgerufener Prozedur werden alle Argumentregister `$a*` und temporäre Arbeitsregister `$t*` vom Stack wiederhergestellt.

# Explizites Sichern von Registern auf dem Stack

Wenn außer den standard-mäßig gesicherten Registern weitere Register gesichert werden sollen, so muss das explizit geschehen. Im Beispiel werden \$t0-\$t2 gesichert.

– **Push:**    `addi $sp, $sp, -12`    # Schaffe Platz für drei Registerwerte auf dem Stack  
              `sw $t0, 8($sp)`  
              `sw $t1, 4($sp)`    } # Sichere Register \$t0-\$t2 auf dem Stack  
              `sw $t2, 0($sp)`



– **Pop:**    `lw $t2, 0($sp)`  
              `lw $t1, 4($sp)`    } # Restauriere Register \$t0-\$t2 vom Stack  
              `lw $t1, 8($sp)`  
              `addi $sp, $sp, 12`    # Lösche Platz von drei Registerwerten auf dem Stack

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

...

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

**2.5.3 Prozeduren**

2.5.3.1 Prinzip: Rücksprungadresse, Parameterübergabe und Stack

2.5.3.2 Realisierung in MIPS

2.5.3.3 Verwendung des Stack

**2.5.3.4 Beispiel: Fakultät**

2.5.3.5 Speicherkonventionen in MIPS

...

# Fakultät als Beispiel

- Das Beispiel Fakultät soll nur eine Übersicht geben, wie eine rekursive Prozedur in Assembler programmiert wird. Die Aufgabenteilung bei der Sicherung von Registern zwischen aufrufender und aufgerufener Prozedur wird erklärt.

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



# Rekursive Prozedur zur Berechnung der Fakultät

- Berechnung der Fakultät von n (also n!)
- Eingabeparameter n steht in \$a0
- Rückgabewert in \$v0

```
int faculty(n)
    if (n<=1) return 1;
    else return n*faculty(n-1);
```

fact:	addi \$sp, \$sp, -8	# adjust stack for 2 items
	sw \$ra, 4(\$sp)	# save return address
	sw \$a0, 0(\$sp)	# save argument
	slti \$t0, \$a0, 1	# test for n < 1
	beq \$t0, \$zero, L1	# if N>=1 goto L1
	addi \$v0, \$zero, 1	# if so, result is 1
	addi \$sp, \$sp, 8	# pop 2 items from stack
	jr \$ra	# and return
L1:	addi \$a0, \$a0, -1	# else decrement n
	jal fact	# recursive call
"\$ra:"	lw \$a0, 0(\$sp)	# restore original n
	lw \$ra, 4(\$sp)	# and return address
	addi \$sp, \$sp, 8	# pop 2 items from stack
	mul \$v0, \$a0, \$v0	# multiply to get result
	jr \$ra	# and return

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

...

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

**2.5.3 Prozeduren**

2.5.3.1 Prinzip: Rücksprungadresse, Parameterübergabe und Stack

2.5.3.2 Realisierung in MIPS

2.5.3.3 Verwendung des Stack

2.5.3.4 Beispiel: Fakultät

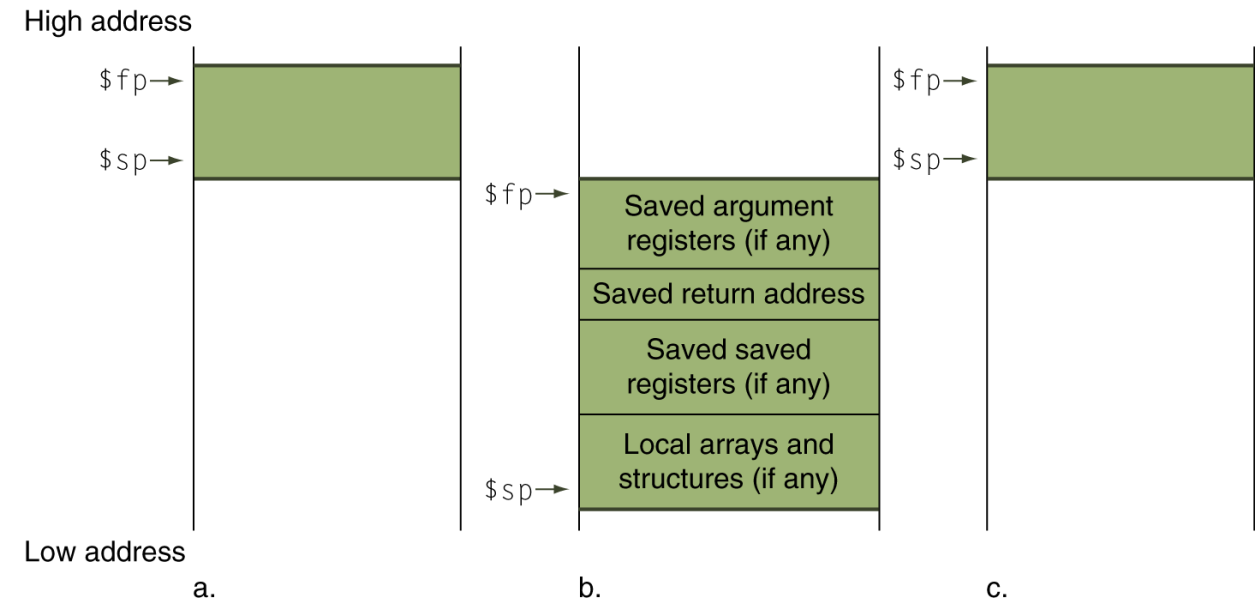
**2.5.3.5 Speicherkonventionen in MIPS**

...

# Aufbau des Procedure-Frames

## Prozedur-Rahmen

- Das Stack-Segment das zu einer Prozedur gehört heißt Procedure-Frame oder Prozedur-Rahmen
- beinhaltet lokale Variablen, gesicherte Register, die Rücksprungadresse und gesicherte Argumente
- befindet sich zwischen Stack-Pointer und Frame-Pointer
- über den Frame-Point lassen sich lokalen Variablen unabhängig vom Stack-Pointer adressieren, was vorteilhaft ist, da dieser im Gegensatz zum Stack-Pointer unverändert bleibt.



- a. Veränderung des Stacks vor Prozeduraufruf
- b. Während Prozedurabarbeitung
- c. Nach Rückkehr aus der Prozedur

# Zusammenfassung: Prozeduraufruf und -rücksprung

## Vor dem Prozeduraufruf in MIPS

1. Sicherung temporärer Register die später benötigt werden
  - `$t0-$t9, $a0-$a3, $v0-$v1`
2. Argumentübergabe
  - die ersten 4 Argumente über Register `$a0-$a3`
  - weitere Argumente über den Stack
3. Prozeduraufruf mittels `jal`-Instruktion

## Am Begin einer Prozedur

1. Stackpointer erniedrigen, um Platz zu reservieren
2. Sicherung aller Register, die vor dem Rücksprung restauriert werden müssen
  - `$s0-$s7, $fp, $ra`
3. Einrichten des Framepointers `$fp`

## Am Ende einer Prozedur

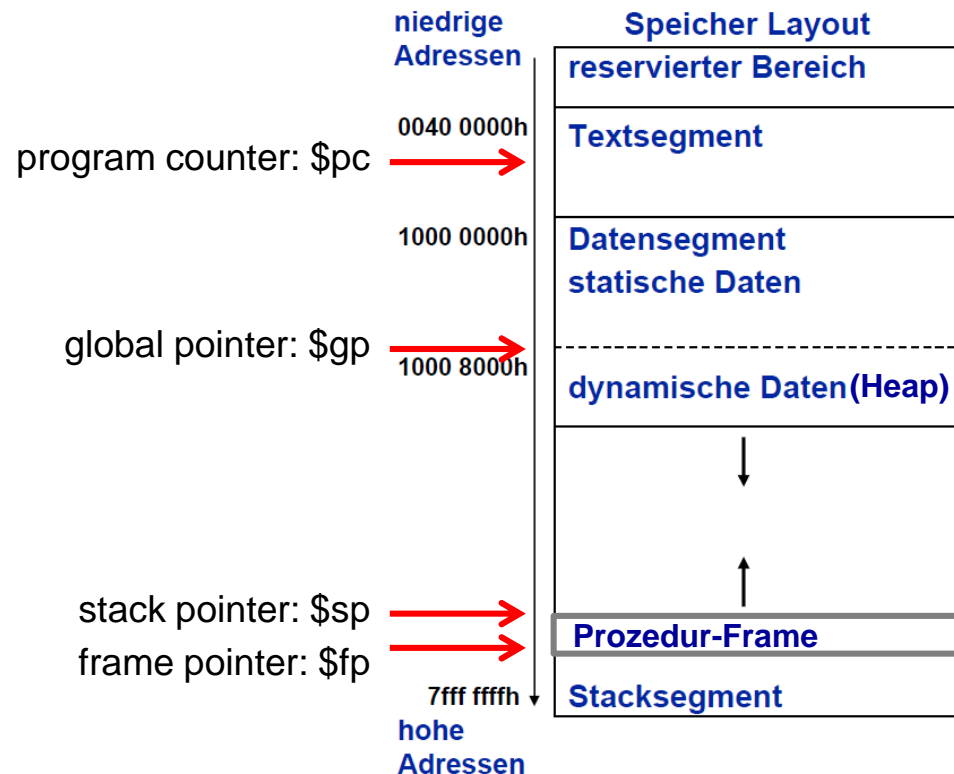
1. Rückgabewerte in `$v0-$v1` speichern
2. Restaurierung aller Register die zu Beginn gesichert wurden
3. Anpassen des Stackpointers `$sp`
4. Rücksprung mit `jr $ra`

## Nach einem Prozeduraufruf

1. Rückspeicherung vorher gesicherter Register
2. Entfernen der Argumente die über den Stack übergeben wurde
3. Anpassen des Stackpointers `$sp`

# Konvention für die Speicheraufteilung in MIPS

- Der Speicher in MIPS ist durchorganisiert. Das Textsegment beinhaltet den Programm Code. Das Datensegment enthält einen Teil für statische Daten wie Konstanten und Variablen fester Größe sowie einen Teil für dynamische Daten, der Heap genannt wird. Dynamische Daten sind alle Daten, für die während der Laufzeit Speicher reserviert wird (malloc, new). Beispiele sind Listen oder Array mit variabler Größe. Das Stack-Segment wird wie in den letzten Folien beschrieben für die Sicherung von Registern bei Prozeduraufrufen sowie für lokale Variablen genutzt.



- **reservierter Bereich**
  - Betriebssystemcode
- **Textsegment**
  - auszuführende Befehlsfolge
  - adressiert über **Program Counter (pc)** (= Befehlszeiger)
- **Datensegment**
  - statische Daten adressiert über „**global pointer**“ **Register \$gp**
  - dynamische Daten
- **Stacksegment**
  - Sicherung von Registerinhalten bei Prozeduraufrufen adressiert über „**stack pointer**“ **Register \$sp**

# Beispiel einer C Funktion

```
int main(){
    int s,i,e;
    int *A,*B;
    s=10;e=20;
    A=(int*) malloc(sizeof(int)*50);
    for (i=0;i<50;i++){
        A[i]=i*i;
    }
    B=fun1(s,e,A);
    return 0;
}
```

- 5 lokale Variablen
  - \$s-Registern ausreichend
- malloc()
  - belegt Speicher im Heap

```
int* fun1(int s, int e, int *A)
{
    int i;
    int *B;
    B=(int*) malloc(sizeof(int)*50);
    for (i=s;i<=e;i++)
    {
        B[i]=i+A[i];
        printf("%d,%d\n",i,* (B+i));
    }
    return B;
}
```

- 3 Argumente
  - \$a Register ausreichend
- 2 lokale Variablen
  - \$s-Register ausreichend
- malloc()
  - belegt Speicher im Heap

# Beispiel einer C Funktion

```
int main(){
    int s,i,e;
    int *A,*B;
    s=10;e=20;
    A=(int*) malloc(sizeof(int)*50);
    for (i=0;i<50;i++){
        A[i]=i*i;
    }
    B=fun2(s,e,A);
    printf("%d", B[0]);
    return 0;
}
```

- Was passiert?
  - **Segmentation Fault!!!**
- **Speicherplatz von B auf dem Stack wird bei Rücksprung wieder frei gegeben**

```
int* fun2(int s, int e, int*A)
{
    int i;
    int B[50];
    for (i=s;i<=e;i++)
    {
        B[i]=A[i];
    }
    return B;
}
```

- eine lokale Variable
  - \$s-Register ausreichend
- lokales Array
  - kann nicht mit Registern realisiert werden
  - aufgerufene Funktion muss im Stack Speicher für Array B (mit fester Größe) belegen

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.3 Prozeduren

2.5.4 **Beispiel: Bubble-Sort**

2.5.4.1 **Bubble-Sort Algorithmus**

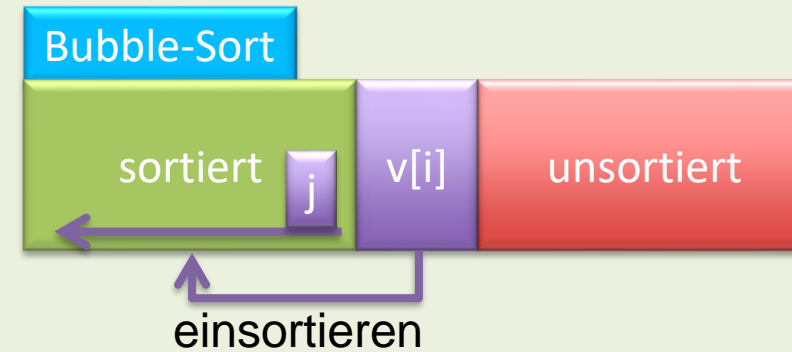
2.5.4.2 Swap in MIPS

2.5.4.3 Sort in MIPS



# Beispiel Bubble-Sort

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```



```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Zunächst: swap

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.3 Prozeduren

2.5.4 **Beispiel: Bubble-Sort**

2.5.4.1 Bubble-Sort Algorithmus

**2.5.4.2 Swap in MIPS**

2.5.4.3 Sort in MIPS

# Prozedur SWAP

## Allgemeine Schritte für Prozeduren

- Zuweisen von Registern für Programm-Variablen (Register planen)
- Code für den Hauptteil der Prozedur produzieren
- Register für den Prozeduraufruf sichern

## SWAP: Register planen

- Aufrufparameter:
  - reserviert sind `$a0-$a3`
  - das reicht `$a0=v` und `$a1=k`
- temporäre Variablen
  - nutzen `$t0` für `temp`

## SWAP: Register für den Prozeduraufruf sichern

- nicht nötig bei einer „Leaf-Prozedur“, in der keine `$s`-Register genutzt werden
  - bei Swap ist die Anzahl `$t`-Register ausreichend
- „Leaf“-Prozedur: kein weiterer Prozeduraufruf

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

# Prozedur SWAP

## C-Code

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

## Assembler-Code

```
swap:
    sll $t1, $a1, 2      # $t1 = k * 4
    add $t1, $a0, $t1    # $t1 = v+(k*4)
                        # (address of v[k])
    lw $t0, 0($t1)       # $t0 (temp) = v[k]
    lw $t2, 4($t1)       # $t2 = v[k+1]
    sw $t2, 0($t1)       # v[k] = $t2 (v[k+1])
    sw $t0, 4($t1)       # v[k+1] = $t0 (temp)
```

## Vorbereitung Assembler Code

Temporäre Register:

\$t0: v[k]      \$a0: v

\$t1: &v[k]    \$a1: k

\$t2: v[k+1]

## Prozedurrückkehr

```
jr $ra                # return to calling
                       # routine
```

**Kapitel 1:** Grundlegende Ideen, Technologien, Komponenten

**Kapitel 2:** Befehle: Die Sprache des Rechners

2.1 Befehlssatz: Was ist das?

2.2 Befehle des MIPS Befehlssatzes

2.3 Darstellungen von Befehlen im Rechner

2.4 Logische Operationen

2.5 Kontrollstrukturen

2.5.1 Programmzähler

2.5.2 IF...THEN...ELSE-Anweisungen und Schleifen

2.5.3 Prozeduren

2.5.4 **Beispiel: Bubble-Sort**

2.5.4.1 Bubble-Sort Algorithmus

2.5.4.2 Swap in MIPS

**2.5.4.3 Sort in MIPS**

# Prozedur SORT

## SORT: Register planen

- Aufrufparameter:
  - reserviert sind `$a0-$a3`
  - das reicht: `$a0=v` und `$a1=n`
- Lokale Variablen
  - nutzen `$s0` für `i` und `$s1` für `j`

Register:

<code>\$s0: i</code>	<code>\$a0: v</code>
<code>\$s1: j</code>	<code>\$a1: n</code>

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j--=1) {
            swap(v, j);
        }
    }
}
```

# Prozedur SORT

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        ...
    }
}
```

Register:

\$s0: i	\$a0: v
\$s1: j	\$a1: n

```
        move $s0, $zero          # i = 0
for1:    slt $t0, $s0, $a1        # $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq $t0, $zero, exit1    # go to exit1 if $s0 ≥ $a1 (i≥n)

        ...

        addi $s0, $s0, 1         # i += 1
        j for1                  # jump to for1
exit1:
```

# Prozedur SORT

## SORT: Register planen

- Inhalte und Adresse des Arrays in Register speichern:
  - nutze `$t2` für Adresse von `v[j]`
  - nutze `$t3` für Inhalt von `v[j]`
  - nutze `$t4` für Inhalt von `v[j+1]`

Register:

<code>\$s0: i</code>	<code>\$a0: v</code>	<code>\$t2: \$v[j]</code>
<code>\$s1: j</code>	<code>\$a1: n</code>	<code>\$t3: v[j]</code>
		<code>\$t4: v[j+1]</code>

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j--=1) {
            swap(v, j);
        }
    }
}
```



# Prozedur SORT

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

```
for2:      addi $s1, $s0, -1      # j = i-1
           slti $t0, $s1, 0      # reg $t0 = 1 if $s1 < 0 (j < 0)
           bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
           sll $t1, $s1, 2       # reg $t1 = j * 4
           add $t2, $a0, $t1     # reg $t2 = v + (j * 4)
           lw  $t3, 0($t2)       # reg $t3 = v[j]
           lw  $t4, 4($t2)       # reg $t4 = v[j + 1]
           slt $t0, $t4, $t3     # reg $t0 = 0 if $t4 ≥ $t3
           beq $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
           ...
           addi $s1, $s1, -1     # j = j-1
           j for2
exit2:
```

Register:

\$s0: i	\$a0: v	\$t2: \$v[j]
\$s1: j	\$a1: n	\$t3: v[j]
		\$t4: v[j+1]

# Prozedur SORT

## Prozeduraufruf SWAP

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
    ...
}
```

```
move $s2, $a0 # copy parameter $a0 into $s2
move $s3, $a1 # copy parameter $a1 into $s3
```

Register sichern

```
move $a0, $s2 # first swap parameter is v
move $a1, $s1 # second swap parameter is j
```

Aufrufparameter

```
jal swap
```

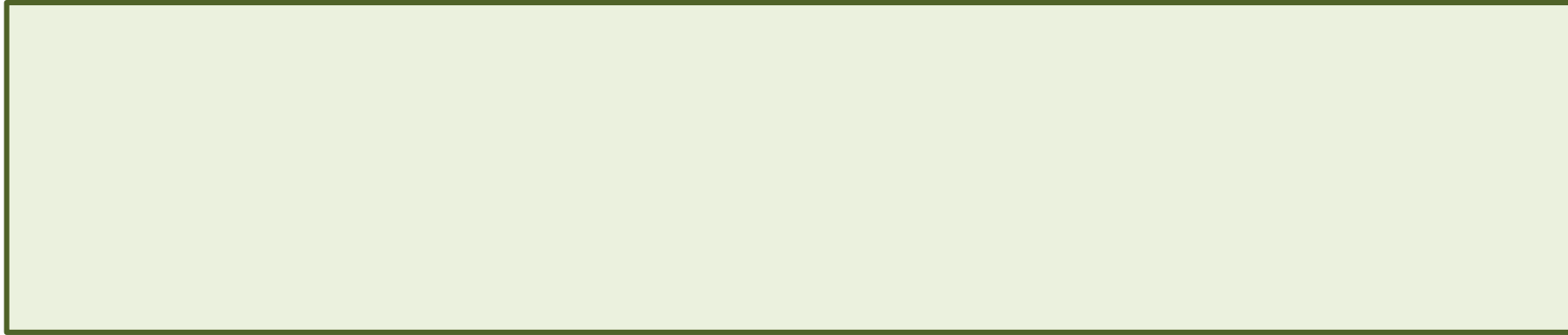
Prozedur aufrufen

Register:

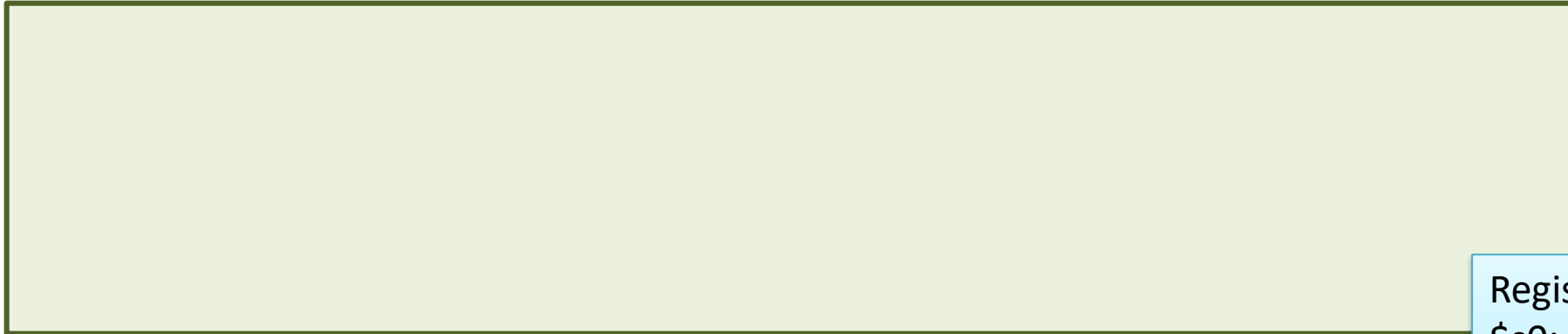
\$s0: i	\$a0: v	\$t2: \$v[j]
\$s1: j	\$a1: n	\$t3: v[j]
\$s2: \$a0		\$t4: v[j+1]
\$s3: \$a1		

# Prozedur SORT

- Register auf Stack sichern



- Register vom Stack wiederherstellen



- Zurückspringen

```
jr $ra                # jump to return address
```

Register:

\$s0: i	\$a0: v	\$t2: \$v[j]
\$s1: j	\$a1: n	\$t3: v[j]
\$s2: \$a0		\$t4: v[j+1]
\$s3: \$a1		

# Prozedur SORT

<Register auf Stack sichern>			
	move \$s2, \$a0	# copy parameter \$a0 into \$s2	Parameter sichern
	move \$s3, \$a1	# copy parameter \$a1 into \$s3	
for1:	move \$s0, \$zero	# i = 0	Äußere Schleife
	slt \$t0, \$s0, \$a1	# \$t0 = 0 if \$s0 ≥ \$a1 (i≥n)	
	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$a1 (i≥n)	
for2:	addi \$s1, \$s0, -1	# j = i-1	Innere Schleife
	slti \$t0, \$s1, 0	# reg \$t0 = 1 if \$s1 < 0 (j < 0)	
	bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
	sll \$t1, \$s1, 2	# reg \$t1 = j * 4	
	add \$t2, \$a0, \$t1	# reg \$t2 = v + (j * 4)	
	lw \$t3, 0(\$t2)	# reg \$t3 = v[j]	
	lw \$t4, 4(\$t2)	# reg \$t4 = v[j + 1]	
	slt \$t0, \$t4, \$t3	# reg \$t0 = 0 if \$t4 ≥ \$t3	
	beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
	move \$a0, \$s2	# first swap parameter is v	Swap aufrufen
	move \$a1, \$s1	# second swap parameter is j	
	jal swap	# zu swap springen	
	addi \$s1, \$s1, -1	# j = j-1	Innere Schleife
	j for2	# jump to for 2	
exit2:	addi \$s0, \$s0, 1	# i += 1	Äußere Schleife
	j for1	# jump to for1	
exit1:	<Register vom Stack wiederherstellen>		
	jr \$ra	# zurückspringen	