
4. Übung

□ MIPS-Assembler

- Pseudobefehle
- Programmiertechniken in Assembler
- Stackprogrammierung
- Unterprogrammaufrufe
- Rekursiver Unterprogrammaufruf



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`



Struktur eines MIPS-Programms

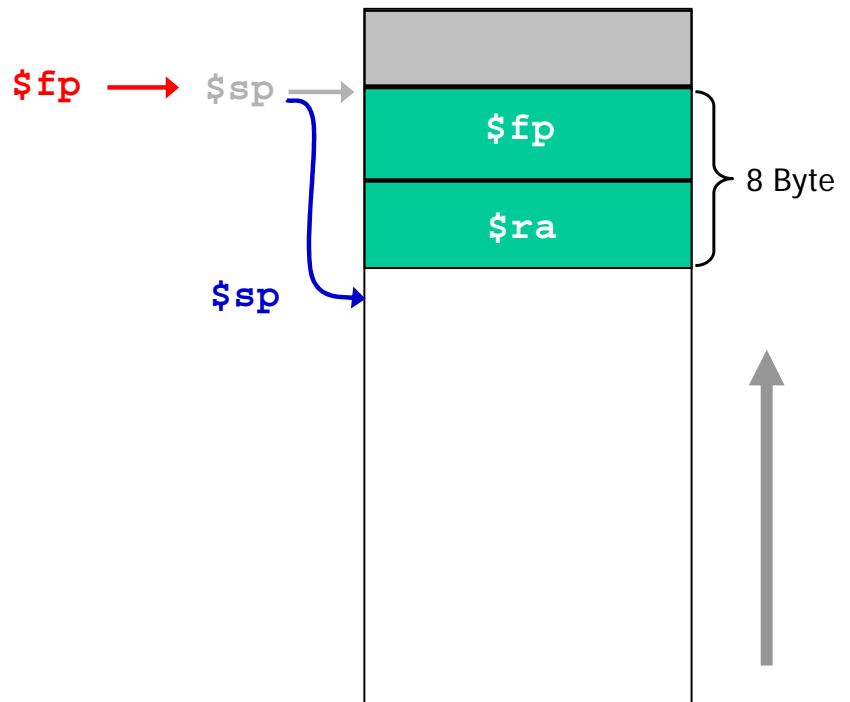
```

.data
# globale Daten
.text
# Unterprogramme

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

      # Hauptprogramm

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



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]</code> <code>ori Rd, \$at, sym & FFFF₁₆</code>
<code>l.d Fd, sym</code>	<code>lui \$at, [sym / 10000₁₆]</code> <code>lwc1 Fd, sym & FFFF₁₆(\$at)</code> <code>lui \$at, [sym / 10000₁₆]</code> <code>lwc1 Fd + 1, sym & FFFF₁₆(\$at)</code>



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



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
```

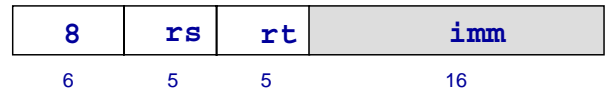


Programmiertechniken

Register \$t1 und \$t2 in den Speicher ab
der Adresse 0x1000004

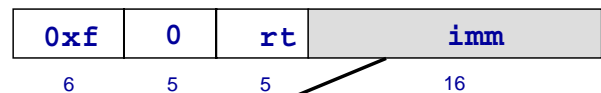
```
addi $s1, $zero, 0x10000004
```

addi rt,rs,imm



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

lui rt,rs,imm



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

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

\$s1



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

16



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

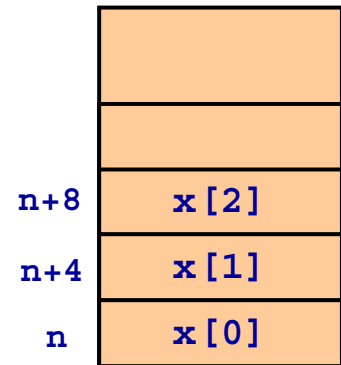
C- Code

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

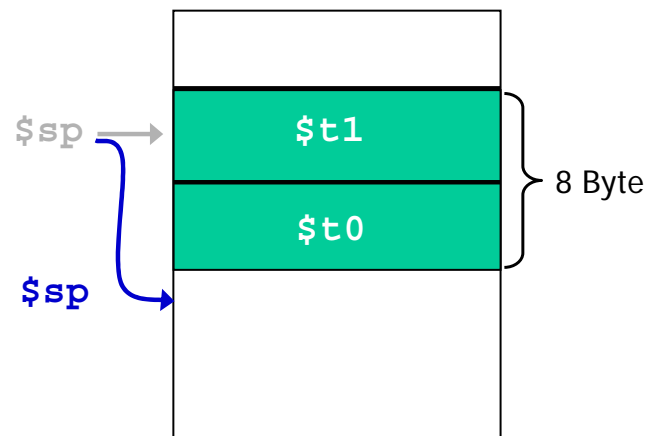
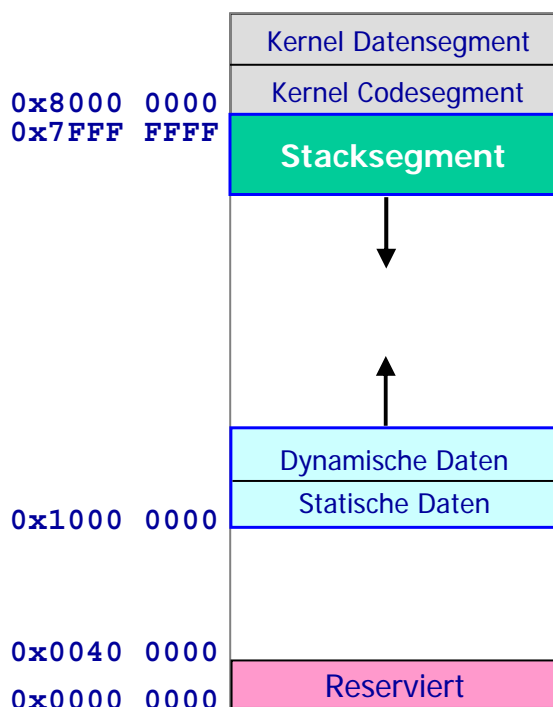
MIPS Assemblercode:

- Annahme: Startadresse n steht bereits in \$t1

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



Stackprogrammierung

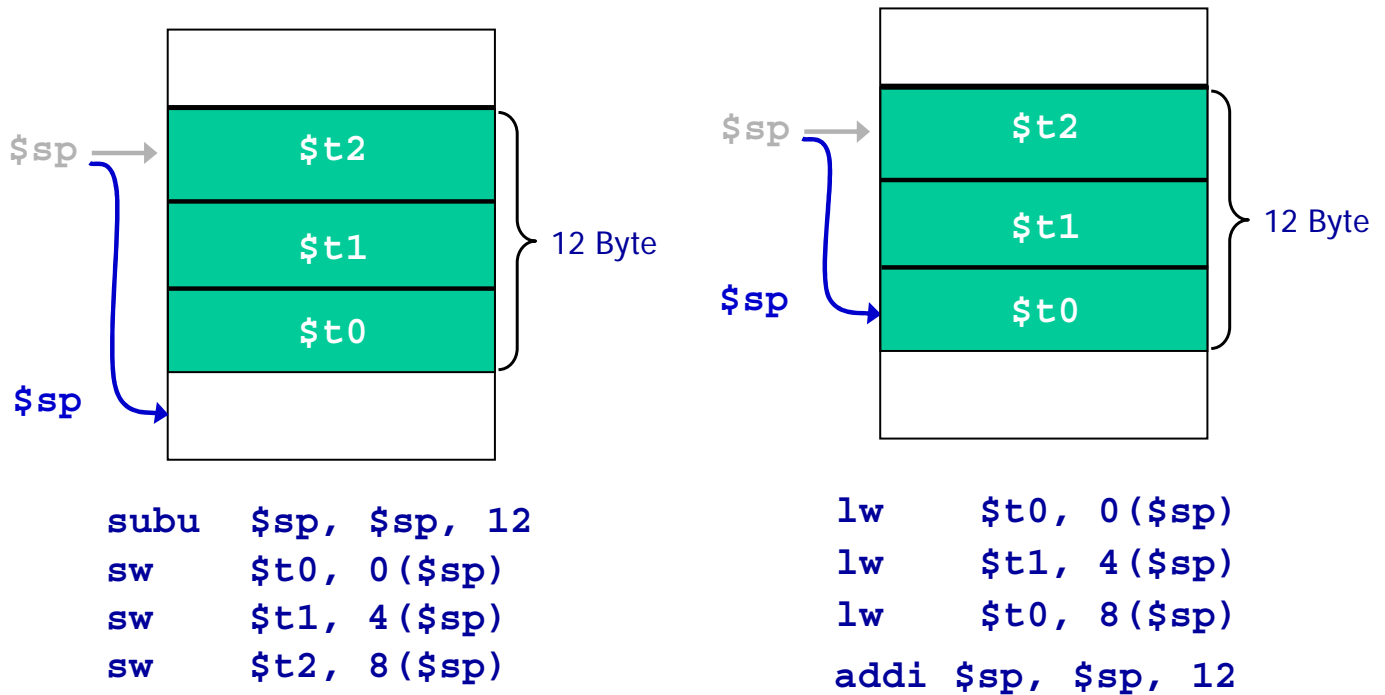


```
subu $sp, $sp, 8
sw $t0, 0($sp)
sw $t1, 4($sp)
```



Stackprogrammierung

□ Lesen vom Stack:



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

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

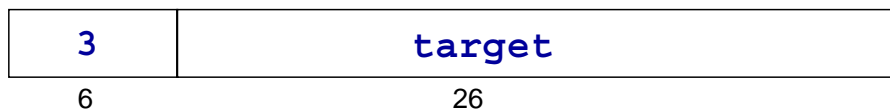


Unterprogrammaufrufe

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

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

```
jal target
```



Springe zu target. Speichere die Adresse des nächsten Befehls in \$ra



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



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
                                           # erfuehlt 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

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



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)



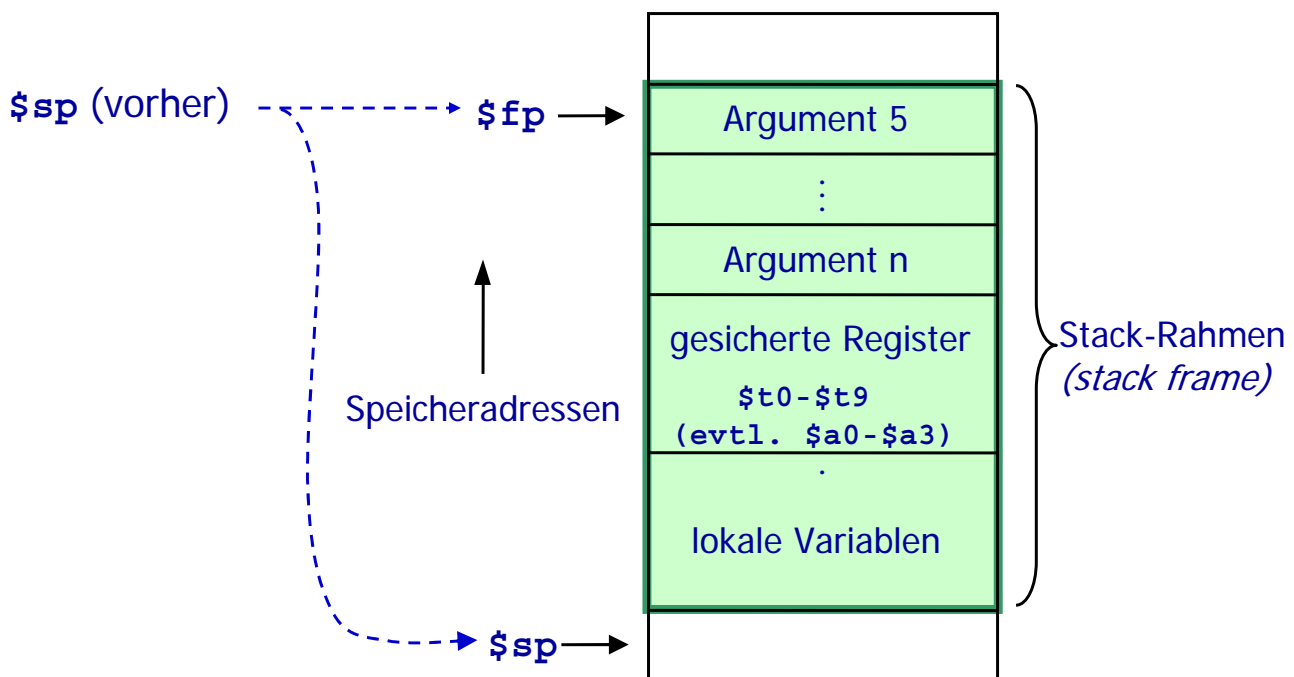
Unterprogrammaufrufe

Aufgaben des Aufrufenden:

- 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 der Rahmenzeigers **\$fp**
- Anlegen eines neuen Rahmenzeigers durch Addition der Rahmengröße zum Stapelzeiger



Stack-Rahmen (*stack frame*)



Einfacher Unterprogrammaufruf

```
main:          jal subroutine
```

```
subroutine:    # keine weiteren
               # Unterprogrammaufrufe
               # für lokale Variablen
               # nur $t0 bis $t9

               jr $ra
```



Beispiel für einen Unterprogrammaufruf

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

               jal subroutine

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

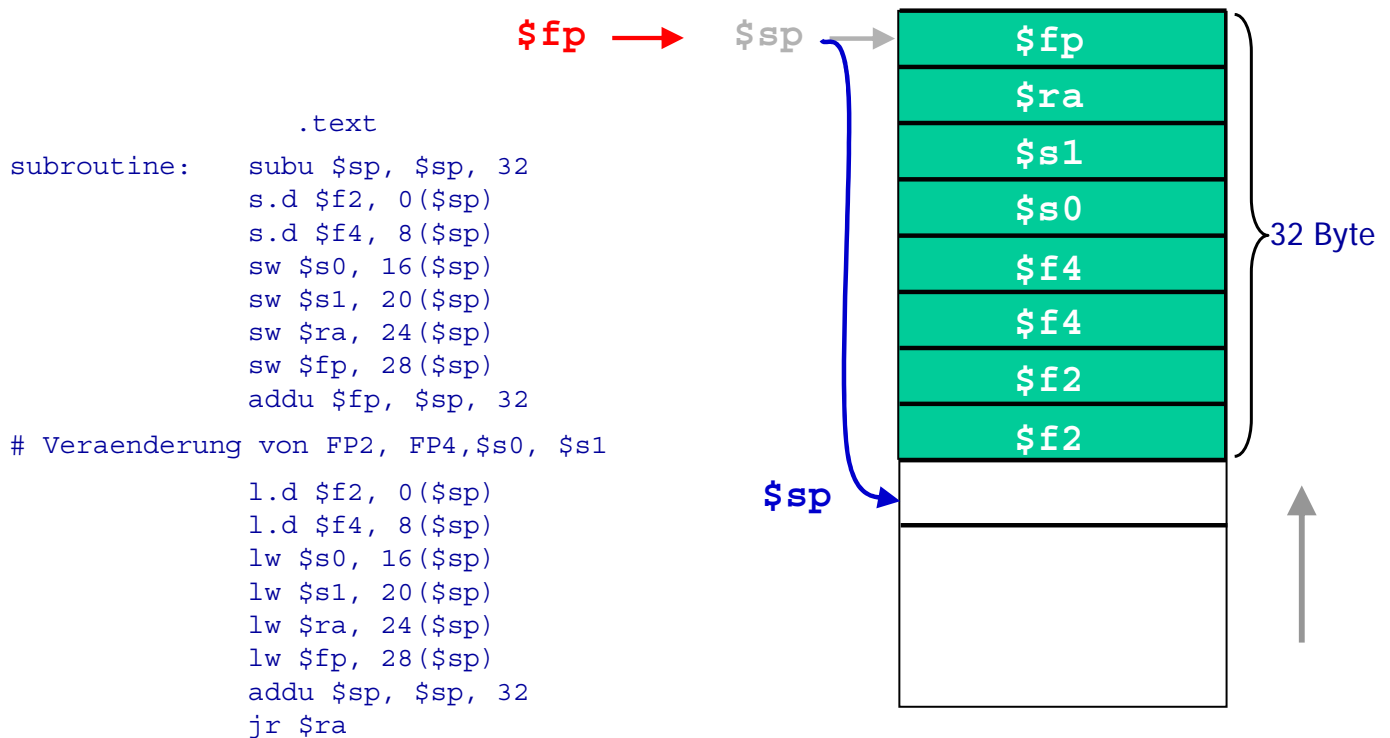
**Unterprogrammaufrufe
mit jal-Befehl !**

```
.text
subroutine:    subu $sp, $sp, 32
               s.d $f2, 0($sp)
               s.d $f4, 8($sp)
               sw $s0, 16($sp)
               sw $s1, 20($sp)
               sw $ra, 24($sp)
               sw $fp, 28($sp)
               addu $fp, $sp, 32

               # Veraenderung von $fp2,$fp4,$s0,$s1
               l.d $f2, 0($sp)
               l.d $f4, 8($sp)
               lw $s0, 16($sp)
               lw $s1, 20($sp)
               lw $ra, 24($sp)
               lw $fp, 28($sp)
               addu $sp, $sp, 32
               jr $ra
```



Beispiel für einen Unterprogrammaufruf



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-Code:

```
fakultaet: subu $sp, $sp, 8          # Stack-Frame von 8 Bytes
            sw $ra, 4($sp)          # Ruecksprungadresse
            sw $a0, 0($sp)          # Argument(n) sichern

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

            add $v0, $zero, 1       # return 1

            lw $ra, 4($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 $a0, 0($sp)       # hole Argument n wieder vom Stack
      lw $ra, 4($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

\$ra (\$fp)	main
\$ra \$a0	fakultaet(10)
\$ra \$a0	fakultaet(9)
\$ra \$a0	fakultaet(8)
\$ra \$a0	fakultaet(7)



Zusammenfassung: Unterprogrammaufruf

Aufrufendes Programm

...
1. Argumente übergeben
2. Temporäre Register sichern
3. Sprungbefehl ausführen

11. Temporäre Register wiederherstellen
...

Unterprogramm

4. Rahmen allokieren
5. Langlebige Register sichern
6. Neuen Rahmenzeiger setzen
...
7. Resultate sichern
8. Langlebige Register wiederherstellen
9. Rahmen deallokieren
10. Springe zurück

